FM TOWNS

FUJITSU FM SERIES PERSONAL COMPUTER

386|ASM TOOL KIT

ユーザーズマニュアル



FUJITSU

FUJITSU

81SP-0073-1

FM TOWNS

386 | ASM TOOL KITユーザーズマニュアル

81SP-0073-1

FM TOWNS

386 | ASM TOOL KIT

ユーザーズマニュアル

富士通株式会社

ごあいさつ

このたびは「FM TOWNS 386 | ASM TOOL KIT V 1.1」をお買い上げいただき、まことにありがとうございます。

本書では、FM TOWNS 上でアセンブラを使用してアプリケーションプログラムを開発するユーザのために、以下のソフトウェアの使用方法を詳細に説明しています。

・アセンブラ	(386 | ASM)
・リンカ	(386 | LINK)
・シンボリックデバッガ	(386 | DEBUG)
・ライブラリアン	(386 | LIB)

本マニュアルには、“外国為替及び外国貿易管理法”に基づく特定技術が含まれています。したがって、本マニュアルまたはその一部を輸出する場合には、同法に基づく許可が必要とされます。

1989年10月

開発キットのマニュアル構成と本書の位置づけ

FMTOWNS 上で、C言語やアセンブラでのソフトウェア開発を可能にするため、富士通では「FMTOWNS 386 | ASM TOOL KIT V1.1」と「FMTOWNS High C™ Compiler V1.4」の２つの関連した開発キットを用意しております。

以下に示すのはこれら２つの開発キットに添付されているマニュアルです。各マニュアルの関連性を考え、本書をお読みください。

製品名称： FMTOWNS 386 | ASM TOOL KIT V1.1

マニュアル	内容
アプリケーション開発キット ユーザーズガイド	開発キットで提供されるソフトウェアやマニュアルの概要、FMTOWNS でアプリケーションを開発する環境や手順についての概要など全体を見渡せる情報が記載されています。
386 | DOS-Extender™ ユーザーズマニュアル	386 | DOS-Extender™のもとで 80386ネイティブモードのプログラムを実行する方法について説明し、さらに 80386ネイティブモード環境についても詳細に解説しています。
386 | ASM TOOL KIT ユーザーズマニュアル	FMTOWNS 上でアセンブラを使用してアプリケーションプログラムを開発するユーザのために、開発の過程で必要な各ソフトウェアの使用方法を詳細に解説しています。 次の４つの部で構成されています。 第１部　アセンブラ　386 | ASM 第２部　リンカ　386 | LINK 第３部　シンボリックデバッガ　386 | DEBUG 第４部　ライブラリアン　386 | LIB

注）網掛けで示されたマニュアルが本書です。

TownsBIOS
リファレンス

FMTOWNS の持つBIOS機能について、リファレンス形式で詳細に解説しています。
BIOSの機能により、次の6つの章に分類されています。

1 グラフィックBIOS　　4 フォントBIOS
2 マウスBIOS　　5 CD-ROM BIOS
3 スプライトBIOS　　6 サウンドBIOS

製品名称： FMTOWNS　High C™ Compiler V1.4

High C™ Compiler
ユーザーズマニュアル

FMTOWNS でHigh C™ Compiler を使用してアプリケーションプログラムを開発するユーザのために、プログラム開発過程、High Cの文法、High Cの標準ライブラリについて解説していまます。
次の3部で構成されています。

第1部　プログラマーズガイド
第2部　ランゲージリファレンス
第3部　標準ライブラリ

Cライブラリ
リファレンス

FMTOWNS 専用のCライブラリについて、その機能や、使用方法、使用例などをリファレンス形式で詳細に解説しています。
ライブラリ関数の機能により、次の8つの章に分類されています。

1 グラフィック　　5 CD-ROM
2 マウス　　6 サウンド
3 スプライト　　7 EUPHONY ドライバ
4 フォント　　8 キーボード

総目次

386｜ASM TOOL KIT
ユーザーズマニュアル

386｜ASM はインテル 80386マイクロプロセッサ用アセンブラです。第１部では386｜ASM でアセンブラソースコードをアセンブルする方法を解説します。

386｜LINKはインテル 80386マイクロプロセッサ用のリンカです。第２部では、386｜LINKを使ってアセンブラまたはコンパイラで作成した別々のオブジェクトモジュールを、単一の80386 アプリケーションに組み合わせる方法を解説します。さらに各種コマンドスイッチをリファレンス形式で解説します。

386｜DEBUG はインテル 80386マイクロプロセッサ用のシンボリックデバッガです。第３部では、386｜DEBUG を使ってプログラムをデバッグする方法を解説します。さらにデバッガコマンドをリファレンス形式で解説します。

386｜LIBは、386｜LINKで生成されたオブジェクトファイルからライブラリファイルを作成するためのライブラリアンプログラムです。第４部では、386｜LIBの使い方と各種コマンドラインスイッチを解説します。

第1部

アセンブラ

386｜ASM

目次

386 | ASM TOOL KIT ユーザーズマニュアル
第１部 アセンブラ 386 | ASM

第8章　アセンブラ制御ディレクティブ　85

図表目次

〈表〉

第1章　概　要

386 | ASM はインテル80386 マイクロプロセッサ用アセンブラです。これはPhar Lap Software, Inc. の80386 ソフトウェアデベロップメントシリーズの一つです。386 | ASM は、１つまたは複数のアセンブリ言語ソースコードファイルを１つのオブジェクトモジュールにアセンブルするために使われます。386 | ASM はまた 8086、8088、80186、80188および80286 マイクロプロセッサ用のプログラムをもアセンブルします。このマニュアルでは、386 | ASM の使用方法を説明します。

386 | ASM は8086、8088、80186、80188または80286 用にコードをアセンブルし、インテル8086オブジェクトモジュールフォーマット（OMF-86）と合致するオブジェクトモジュールを生成します。80386用にコードがアセンブルされると、簡易 OMF-386と呼ばれるOMF-86の単純拡張がオブジェクトファイルフォーマットとして使用されます。簡易OMF-386オブジェクトファイルフォーマットを付録Ｋで説明します。

このマニュアルに使用する符号は以下のとおりです。

{ }	中括弧で囲まれた項目はオプションです。オプション項目が省かれていても文は有効です。
....	前述の項目が任意の回数繰り返されることを示し、項目間はコンマで区切られます。
|	縦線で区切られたいずれか１つの項目だけが使えます。
英小文字	特定の文に適したシンボルに置き換えなければなりません。一般に、使用できるシンボルのクラスをあらわす記述名です。

例えば10バイトのテンポラリ実定数を作るための文の記述は、次の符号で示されます。

```
DT        digits. {digits}  {E {+ | -} digits} ....
```

この１つの実数は、整数部分（必須）と、小数点（必須）と小数部分（オプション）および指数（オプション）（必須文字Ｅ、オプションの＋または－符号、および必須の指数数字からなる）で構成されることを示しています。DT文は、コンマで区切って任意の数の実定数を作成することができます。次の文はいずれも有効です：

```
DT        1.
DT        1.0,1.1,1.2
DT        1.0E2
DT        1.0E+2
DT        1.0E-2
DT        1.E2
DT        0.
DT        0.E0
DT        314.159E-2,0.31459E1
```

このマニュアル中の例では、アセンブラディレクティブと式演算子を示す場合には大文字で記述します。アセンブラ予約語と、ユーザ定義シンボルは小文字を使っています。これは、本書を読み易くするためにしたもので、すべてのアセンブラ予約語とユーザ定義シンボルには、大文字と小文字が使用できます。

第 2 章 386 | ASM

2.1 コマンドラインの形式

アセンブラを実行するためのコマンドラインは、アセンブルタスクイメージの名前（xa386）と、そのあとの、ファイル名とスイッチから構成されます。スイッチは、アセンブラのデフォルトの操作を変更するのに使われます。デフォルトでは、386 | ASM は、1つまたは複数の入力（ソースコード）ファイルをアセンブルして、出力オブジェクトファイルと出力リスティングファイルを作成します。アセンブル中にエラーが発生すると、エラーメッセージが画面に表示され、リスティングファイルに書き込まれます。スイッチを使用すると、画面に表示されているエラーメッセージをエラーリストファイルに転送することができます。

ファイル名には、コマンドラインで拡張子付ファイル名（完全なファイル名）と、拡張子無しファイル名を使用することができます。拡張子の指定がない場合には、386 | ASM が、デフォルトのファイル拡張子を与えます。この時与えられるファイル名拡張子は以下のとおりです。

ファイルタイプ	MS-DOSでのデフォルトの拡張子
入力ソースファイル	.ASM
出力オブジェクトファイル	.OBJ
出力リスティングファイル	.LST
出力エラーリストファイル	.ERR

さらにファイル名を指定する際は、パスを指定することができます。パスの指定がない場合は、カレントデバイスとディレクトリが指定されたものとします。

スイッチはマイナス記号"-"で始まり、この記号の後にスイッチ名を続けます。"-"とスイッチ名の間にはスペースを入れてはいけません。スイッチの引数は、スイッチのすぐあとに置き、スペースで区切ります。入力ファイル名間は、スペースまたはコンマ、またはその両方で区切ります。スイッチ間、または、スイッチと入力ファイル名の間は、スペ

ースで区切らなくてはなりません。入力ファイル名とスイッチは、コマンドラインに任意の順序で配置することができます。386|ASM がファイル名とスイッチを区別するために、入力ファイル名は "-"で始めることができません。有効なコマンドラインの例を次に示します。さらに多くの例を付録Mに示します。

例： 入力ファイルTEST1.ASM2 TEST2.ASMをアセンブルして、オブジェクトファイルTEST1.OBJ リスティングファイルTEST1.LST、およびエラーリストファイルTEST1.ERRを生成します。

```
386asm test1 test2 -errorlist test1
386asm test1,test2 -el test1
386asm -el test1 test1 test2
```

カレントディレクトリ中のTEST1.ASM と、カレントディレクトリのサブディレクトリ(SUBD)中のtest2.Aを入力ファイルにしてアセンブルします。カレントディスクの、トップレベルディレクトリLISTD へリスティングファイルを出力し、OBJD内にオブジェクトファイルを出力します。

```
386asm test1,subd\test.a -list \listd\test1 -o \objd\test1
```

2.2 コマンドラインスイッチ

コマンドラインスイッチは、386|ASM のデフォルトの操作を変更するのに使われます。デフォルトでは、386|ASM は次の作業をします。

- 最初の入力ファイルと同一名を持ち 2.1節に示されたデフォルトファイル拡張子を持ったオブジェクトファイルを作成する。
- 最初の入力ファイルと同一名を持ち 2.1節に示されたデフォルトファイル拡張子と同じ名前を持つリスティングファイルを作成する。
- 80386 をターゲットプロセッサと仮定する。
- 8087または80287 浮動小数点コプロセッサの命令をアセンブルしない。
- ケースインセンティブアセンブリを行う（シンボルの大小文字は、同じに扱われる）。

コマンドラインスイッチは "ー" （マイナス符号）文字で始まり、次にスイッチ名が続きます。スイッチ名には長短の２形式があります。スイッチに対する引数はスイッチ名の直後に置きます。その間に区切り記号としてスペースをおかなければなりません。矛盾するスイッチがコマンドライン上に示された場合は、後の（右側の）スイッチが優先します。

2.2.1　オブジェクトファイルスイッチ

-OBJECTスイッチは、(1) アセンブラにより指定されたデフォルト名以外の名前をオブジェクトファイルに与えます。あるいは(2) オブジェクトファイルをカレントディレクトリ以外のディレクトリに格納します。

形式：　-OBJECT filename

短縮形：-O filename

例：
```
386asm test -o test.o
386asm test -object t.obj
386asm test -o  ¥objdir¥test
386asm test -o  ¥objdir¥test.o
```

－NOOBJECTスイッチは、386｜ASMにオブジェクトファイルを生成しないよう指示します。

形式：　-NOOBJECT

短縮形：-NOO

例：
```
386asm test -noobject
386asm test -noo
```

－NODELETEスイッチは、386｜ASM に、アセンブル中に重大なエラーがあってもOBJ ファイルを削除しないよう指示します。デフォルトでは、重大なエラーが起こると、OBJ ファイルは削除されます。

形式：　-NODELETE

短縮形：-NOD

例：
```
386asm test -noobject
386asm test -nod
```

2.2.2　リスティングファイルスイッチ

－LISTスイッチは、(1)アセンブラにより指定されたデフォルト名以外の名前をリスティングファイルに与えます。あるいは(2)リスティングファイルをカレントディレクトリ以外のディレクトリに格納します。

形式： -LIST filename

短縮形： -L filename

例：
```
386asm test -list test.l
386asm test -l ¥listdir¥test
```

-NOLISTスイッチは386｜ASM にリスティングファイルを生成しないように指示します。

形式： -NOLIST

短縮形：-NOL

例：
```
386asm test -nolist
386asm test -nol
```

-NOSYMスイッチは386｜ASM にソースファイルリスティングの終了時に、シンボルテーブルのサマリーを生成しないよう指示します。

形式： -NOSYM

短縮形：-NOS

例：
```
386asm test -nosym
386asm test -nos
```

2.2.3 エラーリストファイルスイッチ

-ERRORLISTスイッチは、エラーメッセージを格納するファイルを指定します。デフォルトでは、エラーメッセージは、ターミナルに表示されます。エラーメッセージは、エラーリストファイルが生成されても、リスティングファイル（生成されれば）中に必ず格納されます。

形式： -ERRORLIST filename

短縮形：-EL filename

例：
```
386asm test -errorlist test
386asm test -el test.e
```

2.2.4 インクルードサーチディレクトリスイッチ

-INCLUDEスイッチは、1つまたは複数のディレクトリを指定して INCLUDEディレクティブ（3.5 節参照）でインクルードされるファイルを探索するのに使われます。デフォルトでは、カレントディレクトリだけが探索されます。1つまたは複数のインクルードサーチディレクトリがコマンドライン上で指定された場合、386|ASM は、カレントディレクトリより先に、コマンドラインで与えられた順（左から右）に指定されたディレクトリからインクルードファイルを探索します。

コマンドラインに複数の-INCLUDEスイッチを指定するか、単一の-INCLUDEスイッチに複数の引数を与えることによって、複数のインクルードサーチディレクトリを指定できます。単一の-INCLUDEスイッチで多数のサーチディレクトリを指定した場合、サーチディレクトリ間はコンマで区切り、スペースをおいてはいけません。

アセンブラは、ソースファイル中のINCLUDE 文でのファイル名を、-INCLUDEスイッチで指定されたディレクトリに加え、ファイルを見つけようとします。したがって、インクルードディレクトリは、ホストシステム上のファイルパスの指定方法と同じ方法で指定しなければなりません。また、カレントディレクトリからの相対パスも使用されます。

形式：　-INCLUDE dirname {,dirname... }

短縮形：-I dirname {,dirname... }

例：
```
386asm test -include¥includes¥
386asm test -i¥includes¥,isubdir¥
386asm test -i ..¥includes¥
```

参照：　INCLUDE(3.5節)

2.2.5 インストラクションセットスイッチ

インストラクションセットスイッチは、特定プロセッサ（8086、8088、80186、80188、80286、80386）および特定の演算コプロセッサ（なし、8087､80287､80387）用の命令のアセンブルを可能にします。デフォルトでは、80386 プロセッサ用の命令アセンブルが可能ですが演算コプロセッサの命令のアセンブルはできません（したがって、これらのオプションを指定するスイッチは冗長ですが、一貫性のために提供されます）。

特定の命令セットは命令セットディレクティブ（8.2節）を使うことによっても有効になります。ディレクティブは、コマンドラインスイッチに優先します。80286 と80386 の命令セットは、両方とも、そのプロセッサによってだけ特権レベル0で実行される保護命令を含むことに注目してください。そのため、80286 と80386 の命令セットは、非保護命令またはフル命令セット（保護命令も含む）で使用できます。

-8086 スイッチは、8086および8088の命令のアセンブルを可能にします。

形式： 8086

短縮形：86

例： 386asm test -8086
386asm test -86

-80186スイッチは、80186、80188、8086、8088 の各命令のアセンブルを可能にします。

形式： -80186

短縮形：-186

例： 386asm test -80186
386asm test -186

-80286スイッチは、80286 非保護命令と80186、80188、8086、8088 命令のアセンブルを可能にします。

形式： -80286

短縮形：-286

例： 386asm test -80286
386asm test -286

-80286Ｐスイッチは、80286、80186、80188、8086、8088 すべての各命令（保護命令を含む）のアセンブルを可能にします。

形式： -80286Ｐ

短縮形：-286Ｐ

例： 386asm test -80286Ｐ
386asm test -286Ｐ

-80386スイッチは、80386、80286 の非保護命令、および80186、80188、8086、および8088の各命令のアセンブルを可能にします。このモードはアセンブラのデフォルトモードです。

形式： -80386

短縮形：-386

例：
```
386asm test -80386
386asm test -386
```

-80386Ｐスイッチは、すべての80386 命令（非保護命令を含む）のアセンブルを可能にします。

形式： -80386Ｐ

短縮形：-386Ｐ

例：
```
386asm test -80386P
386asm test -386P
```

-8087 スイッチは、8087演算コプロセッサ用の命令のアセンブルを可能にします。

形式： -8087

短縮形：-87

例：
```
386asm test -8087
386asm test -87
```

-80287スイッチは、80287 と8087の演算コプロセッサ用の命令のアセンブルを可能にします。

形式： -80287

短縮形：-287

例：
```
386asm test -80287
386asm test -287
```

-80387スイッチは、80387、80287および8087演算コプロセッサ用の命令のアセンブルを可能にします。

形式：　-80387

短縮形：-387

例：　386asm test -80387
　　　386asm test -387

-N087 スイッチは、演算コプロセッサ用の命令のアセンブルを不可にします。このモードはアセンブラのデフォルトモードです。

形式：　N087

短縮形：N087

例：　386asm test -no87
　　　386asm test -no87

参照：　インストラクションセットディレクティブ（8.2 節）

2.2.6　ケースセンシティビティスイッチ

386 | ASM は、デフォルトでは、ユーザ定義シンボルの大小文字を区別しません。たとえば、386 | ASM はmy_sym, my_SYM, MY_SYMをすべて同じとみなします。386 | ASM に、大小文字を区別してアセンブルさせるために、コマンドラインスイッチが使われます。このオプションを使用すると上の例における３つのシンボルは、すべて異なるシンボルになります。大小文字区別が有効無効にかかわらずアセンブラ内の予約語での大小文字の区別はしません。

-TWOCASEスイッチは、入力ファイルの、大小文字を区別するアセンブルを可能にします。このスイッチを使うと、同じシンボルでも大文字と小文字では、異なるシンボルになります。

形式：　-TWOCASE

短縮形：-TWOC

例：　386asm test -twocase
　　　386asm test -twoc

-ONECASEスイッチは、入力ファイルの、大小文字を区別したアセンブルを不可能にします。このモードはデフォルトモードです。このスイッチは冗長ですが、スイッチの一貫性を保つために設けてあります。

形式：　-ONECASE

短縮形：-ONEC

例：　386asm test -onecase
　　　386asm test -onec

2.2.7　シンボル定義スイッチ

-DEFINE スイッチは、アセンブル時にソースコード中で参照できるテキストシンボルを定義します。この機能は、1組のコードを条件付きアセンブルするために役に立ちます。シンボルの有無はIFDEF およびIFNDEFのディレクティブでテストできます（8.4.3 項と8.4.4 項を参照）。またシンボルの値はIFIDN とIFDIF のディレクティブを使ってテストできます。（8.4.7 項と8.4.8 項参照）。テキストシンボルもまたEQU ディレクティブを使ってソースコード中に生成できることに注意してください。シンボルの定義時に文字列の指定がない場合は、null文字列と等価の値を割り当てます。

形式：　-DEFINE symbol {=string }

短縮形：-D symbol {=string }

例：　386asm test -define cross_mode
　　　386asm test -d opsys=xenix

参照：　IFDEF(8.4.3 項)、IFNDEF(8.4.4 項)、IFIDN(8.4.7項)、IFDIF(8.4.8項)、
　　　EQU(4.2.2 項)

2.2.8　エクストラエラーチェッキングスイッチ

-FULLWARN スイッチは、386|ASM にエクストラエラーチェッキングをさせます。エクストラエラーチェッキング有りの時、特定のセグメントレジスタを使用する命令はあるが、しかしセグメントにそのセグメントレジスタを割り当てるASSUMEディレクティブがないと、386|ASM は、警告します。この警告は、プログラマにセグメントに対する適正なセグメントレジスタポイントを確認させるためです。

シンボルの前方参照によりNOP 命令が生成される時、警告されることがあります。その理由は、アセンブラはパス2上で実際に必要とする以上にパス1上でより多くの領域をリザーブするからです。通常、PTR 演算子で前方参照のシンボルについて多くの情報を386|ASM に与えれば、NOP 命令を排除することができます（9.3.7 項参照）。

2.3 リスティングファイルの説明

386|ASM によって生成されるリスティングファイルは、(1) 入力ファイルからのソースコード行、(2) ソースラインごとのオブジェクトコード、および(3) すべてのユーザ定義シンボル一覧表です。リスティングファイルはページにわけ、読みやすくしています。各ページの頭にページ見出しを印刷します。

リスティングファイルの各行は次のようになります。

```
offset object source
```

この場合offsetとは、セグメントの先頭から生成されたオブジェクトコードまでの16進数のオフセットです。また、objectは、16進数のオブジェクトコードを示し、sourceは入力ファイル中のソースコード行です。たとえば：

```
00000000                    cseg    SEGMENT word public
00000000    B8 00000001             mov eax,1
00000005                    cseg    ENDS
```

ソースコード行がエラーになると、エラーメッセージをリスティングファイルのエラーが発生した行の直後にプリントします。エラーメッセージはそのソースコードファイルとエラー原因となったソースファイル中のソースコード行の番号とを示します。またエラー番号とエラーメッセージも出力します。エラー番号の詳細を、付録Bに記述しています。

リスティングファイルの最後に、プログラム中のユーザ定義シンボルを示すいくつかのシンボルテーブルを出力します。以下でリスティングファイルに現われる順序にこのシンボル表の説明をします。特定のカテゴリー中にシンボルがない場合、そのカテゴリーのシンボル表は出力されません。

2.3.1 ページ見出し

386|ASM は、リスティングファイルの各ページの頭に5行のページ見出しをプリントアウトします。第1行目にはアセンブラのバージョン、アセンブルの日時を表示します。第2行目は空白です。第3行目は、タイトルディレクティブでタイトルの指定がある時は、そのタイトルと、項目番号およびページ番号を表示します。項目番号は、1からスタートし、PAGEディレクティブの制御下で順に増えます。ページ番号は、各項目の始めで1にセットされ、改ページするごとに増えます。ページ見出しの第4行目は SUBTTL ディレクティブでサブタイトルの指定があればそれを表示します。第5行目は空白です。

```
Phar Lap Marco Assembler Version 1.1   Mon Nov 03 09:21:15 1986

Initialization Code                                     Page 1-5
setup_gdt - init global descriptor table

00000000              setup_gdt     PROC        near
00000000 55                          push        ebp
00000001 BB EC                       mov         ebp,esp
                                .
                                .
                                .
```

参照： TITLE (8.3.1項)、SUBTTL (8.3.2項)、PAGE (8.3.3項)

2.3.2 ソースコード文

ソースコード文は、リスティングファイルの１行に次の形式で出力されます：

```
offset object source
```

offsetとは、そのセグメント中のオブジェクトコードの位置を16進法で示すオフセットをいいます。

objectは、ソースコードステートメント用に生成されたオブジェクトコードを示します。オブジェクトコードが生成されない場合は、このフィールドは空白です。普通、オブジェクトコードは、オブジェクトファイルに出力されたのと同一の順序に16進数の文字列（各バイト間はスペースで区切って）として出力されます。しかし、定数またはセグメント内オフセットのようなワードやダブルワード値は、読みやすくするために最上位バイトを先頭に出力されます。最上位バイトを先頭にしたこれらのワードとダブルワード値のバイト間にはスペースをおきません。リスティングファイル中のオブジェクトコードフィールドでは、いくつかの特殊文字が使われます。それぞれの特殊文字とその用法とを表2-1 に示します。

リスティングファイル中のソースフィールドには、ソースコード文が、ソースコードファイル内にあるのとまったく同じ形で含まれています。ソースコードフィールドのすぐ左側には、特殊文字の１つを出力できます。ソースコードフィールドで使用する特殊文字とその意味とを表2-2 に示します。

表2-1　オブジェクトコードフィールドの特殊文字

<table>
<tr><th>文字</th><th>用　法</th><th>説　　明</th></tr>
<tr><td>R</td><td>value R</td><td>value は、リロケータブルである。その最終値はプログラムのリンク時またはロード時に決まる。</td></tr>
<tr><td>E</td><td>value E</td><td>value は、アセンブルされているモジュールに対し外部的に定義される。その値はプログラムのリンク時に決まる。</td></tr>
<tr><td>----</td><td>---- R</td><td>セグメント選択値。この値は、プログラムが実行のためにロードされた時に決まる。</td></tr>
<tr><td>=</td><td>= value</td><td>16進数値value を持つ定数シンボルが生成された。</td></tr>
<tr><td></td><td>= mnemonic</td><td>アセンブラ予約語mnemonicに対する別名シンボルが生成された。</td></tr>
<tr><td></td><td>=</td><td>EQU ディレクティブで生成されたテキスト置換シンボル</td></tr>
<tr><td>:</td><td>xx:</td><td>16進数値xxをもつセグメントオーバーライドプレフィクスバイトを出力した。</td></tr>
<tr><td>/</td><td>xx/</td><td>16進数値xxをもつREP またはLOCKのプレフィクスバイトを出力した。</td></tr>
<tr><td>|</td><td>66 |</td><td>オペランドサイズオーバーライドプレフィクスバイトを出力した。</td></tr>
<tr><td></td><td>67 |</td><td>アドレスサイズオーバーライドプレフィクスバイトを出力した。</td></tr>
<tr><td>[]</td><td>count [value]</td><td>16進数値value がDUP 演算子によってcount 回数複写された。</td></tr>
<tr><td>?</td><td>??</td><td>?オペランドで、1バイト以上の未初期化メモリを確保した。確保した領域の各バイトに対して??が表示される</td></tr>
</table>

表2-2　ソースコード欄における特殊文字

文　字	意　　　　味
I	INCLUDE ディレクティブでインクルードされたファイルからソース文を読んだ。
n	マクロとリピートブロックの展開の深さ(1-9)
+	マクロとリピートブロックの展開の深さが9より大きい。

例：

```
                                                  .LALL
                                                  INCLUDE     macros.asm
                                    I ml          MACRO
                                    I             REPT        2
                                    I             DB          0
                                    I             ENDM
                                    I             ENDM

                                                    ASSUME      cs:cseg
00000000                              cseg          SEGMENT     word public use32

                                                  EXTRN       cval:abs
= 0000000A                            cval2         EQU         10
= 0000000B                            cval3         =           11
= EBP                                 stk_frm       EQU         ebp
=                                     copy          EQU         'Copy'
00000000  0000000A  61 62             bvar          DB          10 dup('ab')
00000014  43 6F 70 79                               DB          copy
00000021  ????                                      DW          ?
                                                    ml
                                      1             REPT        2
                                      1             DB          0
                                      1             ENDM
00000023  00                          2             DB          0
00000024  00                          2             DB          0

00000025  55                                        push        stk_frm
00000026  8B EC                                     mov         stk_frm, esp
00000028  66 | B8 000A                              mov         ax, cval2
0000002C  B8 00000000 E                             mov         eax, cval
```

```
00000031  8D 05 00000000 R                 lea         eax, bvar
00000037  2E: A0 00000000 R                mov         al, bvar
0000003D  66 | B8 ---- R                   mov         ax, cseg
00000041  F3/ A4                           rep movsb

00000043                          cseg     ENDS
```

2.3.3 グループとセグメントのシンボルテーブル

このテーブルはソースコードファイル中で定義したすべてのグループおよびセグメントを表示します。同一グループ内のセグメントは、字下げして表示します。どのグループにも含まれないセグメントは全グループのあとでリストされます。セグメントについてのテーブルの各欄は、セグメントについての使用属性、アライン型、組合せ型、およびクラス文字列を示します。

例： GROUPS AND SEGMENTS

```
Name            Size        Use       Align       Combine     Class

DGROUP. . .     Group
   CONST. .     0000000A    USE32     DWORD       PUBLIC      DATA
   DATA . .     00000258    USE32     DWORD       PUBLIC      DATA
CSEG  . . .     00001560    USE32     WORD        PUBLIC      CODE
STACK . . .     00001000    USE32     DWORD       STACK       STACK
```

参照： SEGMENT(3.7 節)、GROUP(3.9節)

2.3.4 構造体シンボルテーブル

構造体シンボルテーブルは、ソースコードファイルで定義したすべての構造体定義と構造体フィールドを示します（構造体変数は変数シンボルテーブルに示されている）。同一構造体内の各フィールドは、字下げして表示します。各構造体のサイズ（単位はバイト）が示され、各構造体フィールドのバイトサイズ、構造体先頭からのオフセット（単位はバイト）およびデータタイプが、示されます。

例： RUCTURES

```
Name              Size          Offset        Type

PTR386 . . . .    00000006
 SOFFS . . .      00000004      00000000      DWORD
 SELECTOR .       00000002      00000004      WORD
```

参照： 構造体定義（6.6 節）

2.3.5 レコードシンボルテーブル

レコードシンボルテーブルは、ソースコードファイルで定義した全レコード定義と全レコードフィールドを示します（レコード変数は、変数シンボルテーブルに示されています）。同一レコード内の各フィールドは字下げして表示します。レコードおよびレコードフィールドのサイズ（単位はバイト）、マスク値および初期値を示します。各フィールドについてはレコード内のフィールドのオフセットが（単位はビット）表示されます。

例： RECORDS

Name	Size	Offset	Mask	Initial Value
R1	16		000003FF	0000000A
F1 . .	2	8	00000300	00000000
F2 . .	8	0	000000FF	0000000A

参照： RECORD(6.9節)

2.3.6 マクロシンボルテーブル

マクロシンボルテーブルにはソースコードファイルで定義された全マクロを表示します。各表示項目は、マクロ中の形式パラメータ数とソースコード行数です。

例： MACROS

Name	#Params	#Lines
dbl_add...	2	2
dbl_mult..	2	8

参照： マクロ定義（10.2節）

2.3.7 プロシジャシンボルテーブル

プロシジャシンボルテーブルは、ソースコードファイル中で定義された全プロシジャを表示します。各プロシジャ中で定義したすべてのローカル変数、ローカルラベルおよびローカル定数も示します。各プロシジャのデータタイプ(NEAR またはFAR)、セグメント内のプロシジャのオフセット、プロシジャの有効範囲（PUBLICディレクティブで他のモジュールで使用できるようになっていれば"Public"、それ以外は"Private"）およびプロシジャの属するセグメント名を示します。ローカル変数、ローカルラベルまたはローカル定数に対する各欄は、１つの例外を除いては、変数と、ラベルのシンボルテーブル、定数シンボルテーブルに記述された情報と同じです。例外は有効範囲の欄であり、常に、"Local" と指定されます。この"Local" は、定義されたプロシジャ内だけから参照できるローカルシンボルを意味します。

例： PROCEDURES

Name	Type	Offset	Scope	Segment
init-gdt . . .	N PROC	0000000A	Public	cseg
init-idt . . .	N PROC	00001A10	Public	cseg
init-ldt . . .	N PROC	00000420	Public	cseg
init-paget . .	N PROC	00001038	Public	cseg
LOCAL VARIABLES AND LABELS				
#1	N LABEL	00001060	Local	cseg
#2	N LABEL	00001074	Local	cseg
#ecode . .	WORD	000018F0	Local	cseg
LOCAL CONSTANTS				
#arg1 . .	Integer	00000008		
#arg2 . .	Integer	0000000C		
gdt clr . . .	N PROC	00001F08	Prvate	cseg

参照： プロシジャシンボル(5.3節)、Gグローバルシンボル定義（第７章）

2.3.8 変数とラベルのシンボルテーブル

変数とラベルのシンボルテーブルは、ソースコードファイル中で定義されたすべての変数とラベルを表示します。ただし、プロシジャブロックにとってローカルな変数とラベルは除きます。各欄にはラベルまたは変数のデータタイプ、定義されているセグメント中のオフセット、有効範囲（EXTRN ディレクティブで宣言されたなら"Extern"、PUBLICディレクティブで他のモジュールに使用できるようにされたなら"Public"、グローバルシンボルでなければ"Prvate"）、および配置されているセグメント名が示されます。

例： VARIABLES AND LABELS

Name	Type	Offset	Scope	Segment
ecode . . .	DWORD	0000000A	Prvate	dseg
err-exit .	N LABEL	00000256	Extern	cseg
flag1 . . .	Record	0000000C	Prvate	dseg
gdt	struc	00000010	Public	dseg

参照： 変数とデータ定義（第６章）、グローバルシンボル定義（第７章）

2.3.9　定数シンボルテーブル

定数シンボルテーブルは、ソースコードファイル中で定義された（EQU および＝ディレクティブで）すべての定数シンボルを表示します。ただしプロシジャブロックにローカルな定義シンボルは除きます。各定数のタイプは、整定数に対しては"Integer" アセンブラ予約語に対する別名には"Alias" 、テキスト置換シンボルにたいしては"Text"になります。各定数の値も示されます。整定数の場合は、値は、16進数で出力され、シンボルがEXTRNディレクティブで定義された場合には、"Extern"という語が出力されます。さらにその値がPUBLICディレクティブで他のモジュールにも使用できるようになっている場合には、"Public"となります。別名定数の場合は、シンボルは、命令ニーモニック、アセンブラディレクティブ、レジスタ名、式演算子、またはアセンブラキーワードの各々の別名として表示されます。シンボルが表すアセンブラ予約語もまた出力されます。テキスト置換シンボルの場合は、シンボルに変えられたテキスト文字列が示されます。

例：

```
            CONSTANTS
       Name                Type              Value

       cast  . . . .       Text              (pword PTR gdtlim)
       flg_msk  . .        Integer           0000003C
       gdt_size  . .       Integer           Public 00000050
       stk_frame           Alias             Register:EBP
```

参照：　定義定数（第４章）、-DEFINE(2.2.7 項)、グローバルシンボル定義（第７章）

第3章 プログラム構造

3.1 はじめに

この章では386｜ASM でアセンブルされるプログラムの基本構造を定義します。3.2 節ではソースファイルの形式を説明します。3.3 節以降でモジュールネーミング、エントリポイントの仕様、インクルードファイル、プログラムセグメンテーションおよびロケーションカウンタ制御を取扱う各ディレクティブについて説明します。

プログラムセグメンテーションを説明する節では、セグメンテーションディレクティブの使い方について触れます。これは、80386 のセグメントアーキテクチャを説明しようとするものではありません。またプログラムの構築のためにいかにセグメントを使用するかについては説明していません。80386 の概要とそのセグメントアーキテクチャについては、第11章を参照してください。

3.2 386｜ASM ソースファイル形式

アセンブル中、 386｜ASM は、1つ以上のソースファイルから文を読み取り、出力をオブジェクトファイルとリスティングファイルに書き込みます。オブジェクトファイルには、ソースファイル中のステートメントを2進表示し、 386｜LINKが処理できる形式で格納します。リスティングファイルは、アセンブルされた全ソース文を、そのソース文のオブジェクトコード付で示します。

ソースファイルは、ASCII テキストファイルです。このファイルには、内部文字列定数またはテキストの行を区切るコードを除いて、コントロールコードは許されません。有効な文字セットの詳細を付録Cに示します。

386｜ASM によりアセンブルされるソースファイルには、0個以上のソース文があり、そのソース文のあとにEND ディレクティブのあるソース文が続きます。

END ディレクティブがない場合、アセンブラは、最後に処理した入力ファイルの最終行の、そのあとにEND ディレクティブを入れます。 386|ASM では、セグメントと呼ばれる名前付きの領域内にすべての命令バイトとデータバイトを置かなければなりません。通常のソースファイルは、そのEND 文の前に１つ以上のセグメント定義を持ちます。

参照：　形式エレメント（付録C）

3.2.1　文形式

ソースコード文は、ソースファイルに、１行で書きます。文の長さは132 文字以下です。ソース文の形式は：

```
{identifier} mnemonic {operand,...}  {;comment}
```

Identifierはユーザ定義シンボル（オプション）です。Mnemonicは命令ニーモニックまたはディレクティブ名です。このあとにその命令またはディレクティブに対するオペランドのリストが来ます。リストはオプションであり、オペランドをコンマで区切ります。コメント欄はオプションであり、セミコロンで始まります。

例：

```
                PAGE
                DW       0,0        ; Allocate two zero words
    wvar        DW       1          ; Define a word variable
```

3.2.2　シンボル形式

ユーザ定義シンボルは、次の文字を使った文字列です。

```
A-Z
a-z
0-9
_ ? $  @
```

ユーザ定義シンボルの先頭は、数字であってはいけません。シンボル名の最初から31文字だけが有効で、その先はすべて無視されます。 -TWOCASE コマンドラインスイッチで大小文字の識別を有効にしないかぎり、シンボルの大文字と小文字は同一に取扱われます。

例：

```
sym0        DB        ?
_sym?1      DW        ?
@sym_2      SEGMENT   word public use32
$1:
```

386 | ASM は、プロシジャ内のローカルシンボルもサポートします。ローカルシンボルを定義するには、"#" 文字をシンボル名の先頭に置きます。シンボル名中の残りの文字は、通常のシンボル形式の規則に従います。"#" 文字は、シンボル名の先頭以外の位置で使ってはいけません。ローカルシンボルは、それが定義されたプロシジャの範囲内だけで有効となります。したがって、ローカルシンボル名は、異なるプロシジャブロックの中では二重定義シンボルエラーになることなく再使用できます。

例：

```
add_dbl     PROC    near                ; add two doublewords

#arg1       EQU     8                   ; 1st argument
#arg2       EQU     12                  ; 2nd argument

            push    ebp                 ; set up
            mov     ebp,esp             ; stack frame

            mov     eax,#arg1 [ebp ]    ; add two
            add     eax,#arg2 [ebp ]    ; arguments
            jo      #error              ; set error if overflow
#exit:
            pop     ebp                 ; restore original EBP
            ret
#error      call    ovf_err             ; call   error routine
            jmp     #exit

add_dbl     ENDP
```

参照： -TWOCASE(2.2.6項)、プロシジャブロック(5.3節)

3.2.3 定数

386 | ASM は、8086/8088/80186/80188/80286/80386 および8087/80287コプロセッサによりサポートされるすべてのデータタイプと形式をサポートします。整数は、2の補数で表され、その長さは、1、2、4、8、または10バイトです。実数は、80287 浮動小数点形式で表され、その長さは4、8または10バイトです。パック10進数は、80287 パック10進形式で表され、その長さは常に10バイトです。各データタイプによって表される数値の範囲を付録Gに示します。すべての数字は最下位バイトを先頭にしてメモリに記憶されます。80386 データ形式の詳しい説明については、リファレンス6を参照してください。また、8087/80287データ形式については、リファレンス8を参照してください。

参照： データ宣言ディテクティブ（6.2 節)、データタイプ（付録G）

(1) 整定数

形式： digits {radix_specifier}

整数は、１つ以上のdigitsの列で、あとにオプションで１文字の基数指定子が続きます。Digitsは0-9 、16進数の場合はA-F 、またはa-f 文字も使えます。radix_specifierがない場合、その数字の基数は、カレントのデフォルト基数であるとみなします（基数２から基数16までの任意の基数）。デフォルトの基数は、初期状態では10です。しかし、RADIXディレクティブを使って変更できる。基数指定子を表3-1 に示します。

表3-1　基数指定子

基数指定子	基　　数
Ｂかｂ	２進（基数２）
Ｑかｑ	８進（基数８）
Ｏかｏ	８進（基数８）
Ｄかｄ	10進（基数10）
Ｈかｈ	16進（基数16）

すべての整数は、基数に関係なく10進の数字(0-9) で始めます。英字で始めると数字ではなくすべてシンボルとして扱われます。必要があれば０で始まる文字列が使用できます。下記の例はすべて10進の12をそれぞれ異なった基数で指定しています。

例：
```
        DW       12        ; 10進
        DW       0Ch       ; 16進
        .RADIX   16        ; 省略時を16進にセット
        DB       0C        ; 16進12
        DD       14Q       ; 8 進
        DQ       1100B     ; 2 進
        DT       12d       ; 10進
        .RADIX   11        ; 省略時を基数11にセット
        DW       11        ; 基数11
        DW       12D       ; 10進
```

整数は普通、２の補数形式で記憶されます。この整数は80386 または80287 コプロセッサによって処理できます。DTディレクティブは、8087/80287パック10進形式で記憶されるパック10進数を生成するのに使用されます。DTディレクティブはデフォルトの基数が10進で、数字の最後に基数指定子がない時は、パック10進数を生成します。それ以外は、普通の２の補数整数が生成されます。パック10進数の最大桁は、18桁です。

```
例：    .RADIX      10
        DT          12345      ; パック10進数
        DT          12345D     ; 2の補数整数
        DT          12345Q     ; 2の補数整数
        DT          0ABCDH     ; 2の補数整数
```

参照： .RADIX(8.6.2項)

(2) 実定数

形式： digits. {digits} {E {+|-} digits}

実数は、整数部分（必須）、小数点（必須）、小数部分（オプション）および指数（オプション）で示されます。指数は省略されない時は、文字“E”で始まり、次に＋または－符号（オプション）と数字（必須）があります。すべてのdigitsは10進数（0-9)でなければなりません。実数は常に DD、DQ およびDTのデータ宣言ディレクティブと共に使われます。実数は、8087/80287浮動小数点形式で記憶されます。

```
例：  DD     1.0
      DQ     1.0E20
      DT     1
      DD     0.002E-10
      DD     1.0E+20
```

(3) コード化された実数

形式： digitsR

実数はコード形式でも入力できます。プログラマは、8087/80287演算コプロセッサ（または他の演算コプロセッサまたはソフトウェア浮動小数点パッケージ）の使用する浮動小数点形式に実数をコード化します。コード化された実数は、16進数digitsの文字列で示され、そのあとに実数指定子R が続きます。コード化された実数は、常にDD、DQ、DTのディレクティブと共に使用され、桁数はデータタイプのサイズとまったく同じにします。数字の先頭が０である場合は、桁数は１つだけ増してもよいが、そうしなくてもかまいません。このようにDDディレクティブで８桁、DQディレクティブでは16桁、DTディレクティブでは20桁（または先頭が０である場合は、それぞれ9,17,21 ディジット）を使用することができます。

```
例：  DD    1.0E1
      DD    41200000R
      DD    -0.000123
      DD    0B900F98FR

      DQ    1.0E38
      DQ    47D2CED32A16A1B1r

      DT    1.0E1
      DT    4002A000000000000000R
```

(4) 文字列定数

```
形式： "string"
       'string'
```

文字列定数は、ダブル引用符（"）またはシングル引用符(')で囲まれた1文字以上の文字列です。stringの文字は、プリント可能なASCII 文字、NULL(0)を除くプリント不可能なASCII 文字、または非ASCII 文字（最上位ビットを持つ文字のセット）のうちならどれでもかまいません。これにより、グラフィック文字は、コード化して直接文字列の中に置くことができます。

文字列を囲む引用符を、その文字列の中の一文字としてコード化するには、引用符文字を２回タイプします。

文字列をメモリに入れるためには、文字列定数を、DBディレクティブといっしょに使用します。８文字までの文字列なら、指定のデータタイプからオーバーフローしないかぎり、整数値として使用できます。文字列中の各文字のASCII 値は１バイトの整数として右側の文字列が下位のバイトになるように格納されます。

```
例：  DB    'this is a string'
      DB    "don't"
      DB    'don''t'
      DB    1000 DUP ("STACK")

      ;  string encoded integers

      DW    'ab'
      DD    'abcd'
      DQ      DQ      'abcdefgh'
      DW      DW      'ab'DUP(?)
```

3.3 NAME ディレクティブ

形式： NAME modname

アセンブラによって出力されたオブジェクトファイルには、アセンブルされたモジュール名をもつ見出しレコードがあります。386|LINKは、モジュールの参照が必要な時に、この名前を使います。

このNAMEディレクティブは、このモジュール名をオブジェクト出力ファイルに設定するのに使われます。modname パラメータは、任意の長さの文字列で、頭の132 文字だけが有効です。アセンブル中の1モジュールにつき1個のNAMEディレクティブだけが使用できます。モジュールに複数のNAMEディレクティブがある場合、名前を変えようとするたびにエラーとなり、最初のNAMEディレクティブによって指定された名前が使われます。

アセンブル中のモジュール名の指定がない場合は、アセンブラは、次の規則でデフォルト名を定めます：

1) リスティングファイルページのタイトルをTITLE ディレクティブで指定している場合、タイトル文字列の頭の6文字（文字列が6文字より短い場合は、6文字未満）がモジュール名になります。
2) モジュールに名前がなくタイトルの指定もない場合、アセンブラは、ソースコードファイル名をモジュール名に指定します。

例： 次の例が、モジュール名mod1の生成例です。

```
TITLE This is the first module of the system
NAME mod1
END
```

名前の指定がない場合、タイトルの頭の5文字を使います。モジュール名MTH386の生成例を次に示します。

```
TITLE MTH386 - An 80386 long arithmetic library
END
```

参照： TITLE(8.3.1 項)

3.4 ENDディレクティブ

形式： END {entry_pt}

END ディレクティブは、カレントモジュールのアセンブルを終了させ、そのプログラムのオプションのエントリポイントを指定します。アセンブラはEND ディレクティブに出会

うと、入力ファイルの終りであるとみなします。そのディレクティブに続くテキストは無視され、リスティングファイルにコピーされません。アセンブラがINCLUDE ディレクティブでインクルードされたファイルの処理中にEND 文に出会った場合も、結果は同じです。すべての入力ファイルはクローズされ、アセンブルは終了します。

エントリポイント指定子entry_pt（オプション）は、実行開始時に80386 インストラクションポインタにロードする数値を指定します。このエントリポイントは、モジュール中のアドレスを求める任意の式です。プログラム中の１個のモジュールだけがエントリポイントを指定できます。それ以外は、プログラムのリンク時にエラーになります。

例：　次に示すEND ディレクティブは、モジュールのアセンブルを終了させ、プログラムのエントリポイントとしてのプロシジャstart を確立します。

```
TITLE Example of entry point specification with END

code    SEGMENT word public

start   PROC far
        mov eax,1   ; This will be the first instruction

start   ENDP

code    ENDS

END     start
```

3.5 INCLUDEディレクティブ

形式：　INCLUDE filename

INCLUDE ディレクティブは，別ファイルに格納してあるテキストをアセンブル中のファイルに挿入します。インクルードファイルには、通常、システムのデータ構造、グローバル変数およびパブリックプロシジャの定義が含まれています。インクルードファイルは、任意の深さにネストできます。

filename引数は、アセンブル中にインクルードされるファイル名を指定します。アセンブラはまず、コマンドライン上で指定されたインクルードライブラリディレクトリ名にfilenameを加えてファイルをオープンします。これに失敗すると、アセンブラは指定されたファイル名でファイルをオープンします。

例：　次のプログラム例は、ヘッダファイルからデータ構造定義と外部関数定義をインクルードします。

```
TITLE Example of using INCLUDE
INCLUDE dstruct.h
INCLUDE extern.h

code     SEGMENT word public

start    PROC far

         mov   ax.field1 [ebx]    ;field1 is defined in
                                  ;dstruct.h
         call print_field         ;print_field is defined in
                                  ;extern.h
         ret

start    ENDP

code     ENDS
END
```

参照：　LISTI(8.3.6 項)、XLISTI(8.3.7 項)、-INCLUDE(2.2.4 項)

3.6 COMMENTディレクティブ

形式：　COMMENT delim text delim

COMMENT ディレクティブを使うと、アセンブラは区切り文字delim 間のtextをコメントとして扱います。コメントは、リスティングファイルにコピーされるが、アセンブラによって処理はされません。COMMENT ディレクティブのあとの最初の非余白文字は、コメントの区切り文字です。うしろの区切り文字は、COMMENT ディレクティブと同じ行におく必要はありません。これによって複数行のコメントが可能になります。後の区切り文字と同一行にある残りのテキストも、コメントとして扱われます。

例：　複数行コメントの例を以下に示します。

```
TITLE An example of a multi-line comment
COMMENT *
```

```
        The text contained on these lines will be treated
        as a comment because the second occurrence of the
        delimiter does not occur until below.

      * End of COMMENT
      END
```

3.7 SEGMENTディレクティブ

形式： segname SEGMENT {align} {combine} {useatr} {access}
{'class'}

SEGMENT ディレクティブは、アセンブル中のモジュールの論理部分の始めをマークします。アセンブラによって生成されたすべての命令およびデータバイトは、ENDSディレクティブがそのセグメント用に処理されるまで segnameという名のセグメント中に置かれます。プログラムモジュールがリンカによってリンクされると、同一名のすべてのセグメントは、連結されリロケートされます。

segname パラメータは、オープン中のセグメント名を指定します。ディレクティブのあとにオプションパラメータを置くと、セグメントの使用法に関する情報をリンカとローダに与えます。これらオプションパラメータは、どんな順序でも指定できます。アセンブラは各パラメータがどのタイプであるかをその値に基づいて決めます。パラメータの指定がない場合は、386|ASM によってデフォルト値が選択されます。デフォルト値を選択する規則は、パラメータにより異なり、各パラメータを記述する部分で指定されます。

386|ASM は、アセンブル時にセグメントを複数回オープンしたりクローズしたりすることができます。また、同一名のすべてのセグメントを単一セグメントとして扱います。セグメントの再オープン時、ロケーションカウンタは、そのセグメントを最後にクローズした時の値でスタートします。しかし、再オープン中のセグメント用のセグメント属性指定子（オプション）は、そのセグメントを最初にオープンした時に指定した指定子と合致していなければなりません。

セグメント定義はネストできます。あるセグメントｓa1を別のセグメントsa0 内でオープンすると、セグメントsa0 へのアセンブルは、一時的に中止されます。ネストされたセグメントsa1 内の文によって発生するすべてのデータは、そのセグメントの中に置かれ、セグメントsa0 用のロケーションカウンタは、変更されません。ENDSディレクティブが、ネストされたセグメントsa1 で処理されると、アセンブリはセグメントsa0 へ移ります。セグメント定義のネストは可能ですが、これらの定義はオーバーラップできません。ネストされたセグメントsa1 のENDSディレクティブは、セグメントsa0 のENDSディレクティブに先行します。先行していないと、386|ASM はエラーを通知します。

各セグメント属性指定子（オプション）、それらがとる値およびその値の意味を以下の項で説明します。

参照： ENDS(3.8節)、GROUP(3.9節)、ASSUME(3.10節)

3.7.1 アライン型指定子

セグメントのアライン型は、セグメントのベースアドレスとしてどの値を使用できるかをリンカに伝えます。386|LINKがオブジェクトモジュールを結合して実行可能なイメージを生成する時、必要ならセグメント間のスペースをnullバイトで埋め込み、セグメントの各部分が、そのアライメント型とコンパチブルなベースアドレスをもつことが保証されます。アライン型指定子は、次の値のいずれかをとります：

アライン型	意　　　味
BYTE	セグメントは、任意のアドレスに置くことができる。
WORD	セグメントはワード境界になければならない（ベースアドレスは偶数である）。
DWORD	セグメントはダブルワード境界になければならない（ベースアドレスは4の倍数である）。
PARA	セグメントはパラグラフ境界になければならない（ベースアドレスは16の倍数である）。
PAGE	セグメントはページ境界になければならない（ベースアドレスは256 の倍数である）。
PAGE4K	セグメントは4キロバイトページ境界になければならない（ベースアドレスは4096の倍数である）。

セグメントに対するアライン型の指定がない場合は、デフォルト値としてPARAが使われます。

3.7.2 組合せ型指定子

セグメントの組合せ型は、同一名のセグメント同士を組み合わせる方法をリンカに伝えます。組合せ型指定子は、以下のいずれかの値を持ちます。

組合せ型	意　　　味
PUBLIC	PUBLIC組合せ型を持つ同一名のセグメントは、リンカによって連結されて１つのセグメントを作ります。セグメントの各部分は、結合時に新しいセグメントベース相対でリロケートされます。
STACK	STACK 組合せ型を持つ同一名のセグメントは、PUBLIC　組合せ型を持っているかのように、リンカによって連結されます。唯一の相違点は、SS:ESPの初期化をサポートする実行可能タスクイメージ形式に対して、リンカは、SSおよびESP のレジスタの初期値をセグメントの終りを指すようにセットすることです。
COMMON	COMMON組合せ型を持つ同一名のセグメントは、オーバーレイされます。それぞれの構成セグメントは、同一アドレスつまり、複合セグメントのベースにリロケートされます。複合セグメントの長さは、最も長い構成セグメントの長さです。COMMONセグメントは、初期化されたデータを持つことができますが、COMMONセグメント中の同一領域を別々のモジュールの別々の値に初期化しないよう注意してください。
MEMORY	MEMORY組合せ型は、PUBLIC組合せ型と同義語です。この型は、インテル互換性を備えています。
AT address	AT指定子は、プログラムの実行時、セグメントがパラグラフベースのaddress を持つということをアセンブラとリンカに知らせます。addressパラメータには、絶対アドレスを表す任意の式が許されます。ATセグメントは、PROMに焼き付けるプログラムで使用します。これらのプログラムは、通常インテルの16進形式などの絶対形式でリンクされます。リロケータブルなタスクイメージ（たとえばMS-DOS.EXE形式）でリンクされるプログラムについては、ATセグメントに初期化データを含めることはできず、既にメモリ中にあるデータのテンプレートとしてだけ役立ちます。 …………………………………… セグメントに対してAT組合せ型を指定する場合、ライン　上の最後のセグメント属性指定子で指定します。また、　AT組合せ型はPARAアライン型を意味することにも注意し　てください。このため、AT組合せ型で定義されたセグメ　ントについては、アライン型を指定できません。 …………………………………… セグメントの定義時に、組合せ型の指定がない場合は、デフォルトのプライベート組合せ型となりそのセグメントは他のセグメントとは組み合わされません。

3.7.3 使用属性指定子

セグメントの使用属性指定子は、セグメントのディスクリプタテーブル（リファレンス6参照）にあるDビットの実行中の期待値をアセンブラに伝えます。80386 CPU は386 ネイティブモードにおいて、デフォルトのオペランドおよびアドレスモードのサイズが16ビットか32ビットかを決めるのにDビットを使います。

アセンブラは、アドレスサイズとオペランドサイズのオーバーライドバイトを正しく生成するために、プログラム実行時にDビットがどの値になるかを知る必要があります。使用属性指定子は、次の２値のうち１つを持ちます。

使用属性	意　　　味
USE16	生成中のセグメントのデフォルトのオペランドサイズおよびアドレスサイズは16ビットです。命令およびデータを参照するためのすべてのオフセットはデフォルト16ビットになります。命令オペランドのデフォルトサイズは16ビットです。生成中のセグメントの最大サイズは、64Kバイトに制限されます。
USE32	生成中のセグメントのデフォルトのオペランドサイズおよびアドレスサイズは32ビットです。命令およびデータを参照するためのすべてのオフセットはデフォルトで32ビットになります。命令オペランドのデフォルトサイズは32ビットです。

セグメント使用属性の指定がない場合は、そのセグメントをネストしているセグメントの使用属性とアセンブルのターゲットCPU とに基づいて、デフォルトの値が割り当てられます。セグメントが別セグメント内にネストされる場合、そのセグメントをネストするセグメントの使用属性が与えられます。ネストされていない場合は、ターゲットCPU が80386 であればデフォルト値としてUSE32 を割り当てられます。それ以外は、USE16 が割り当てられます。

8086/8088/80186/80188/80286 プロセッサまたは、80386 のリアルモードで実行するプログラムは、USE16 セグメントだけを持つことができます。80386 の386 ネイティブモードで実行するプログラムは、USE16 またはUSE32 のいずれかをもつことができます。

参照：　第11章、付録J

3.7.4　アクセス型指定子

アクセス型指定子は、プログラム実行時にセグメントにアクセスする方法を定義します。この指定子は次のいずれかの値を持ちます。

アクセス型	意　　　味
RO	セグメントは、読取り専用のデータセグメントである。
EO	セグメントは、実行専用のコードセグメントである。
ER	セグメントは、読取り可能なコードセグメントである。
RW	セグメントは、書込み可能なデータセグメントである。

セグメントに対して、アクセス型の指定がない場合、デフォルト値としてERが割当てられます。ターゲットCPU が80286 または80386 の場合に限りアクセス型を指定します。8086、8088、80186、80188のCPU はセグメントプロテクションをサポートしません。

3.7.5　クラス指定子

セグメントのクラス指定子は、そのセグメントがメンバとなっているクラス名を定義します。クラス名は、シングル引用符で囲まれ、その長さは任意です。セグメントにクラス指定子が指定されていない場合は、デフォルト値としてnullクラス名が割り当てられます。

386 | LINKは、メモリ中のセグメントの配置決定のアルゴリズムの一部としてセグメントのクラス名を使います。同一クラス名のセグメントは隣接したメモリに配置されます。セグメントがnullクラスのメンバである場合には、やはりnullクラスメンバに属する他のセグメントと隣接して配置されます。

3.8　ENDS ディレクティブ

形式：　segname ENDS

ENDSディレクティブは、segname の現定義を終了させます。そしてそのセグメントをネストしているセグメントがあるときは、そのセグメントにアセンブルを移します。そのセグメント用のカレントロケーションカウンター値は保存され、セグメントの再オープン時にはアセンブルをそこから再び続行します。

segname パラメータは、最後にオープンしたセグメント名でなければなりません。異なるセグメント名であると、ブロックネスティングエラーとなり、セグメントはクローズされません。

ENDSディレクティブは構造体定義を終了させるためにも使います（6.6.2 項参照）。アセンブラは、segname パラメータを使ってENDSをどのように使用するのかを決めます。

例：　8086ターゲットCPU 用にアセンブルされているモジュールのために生成したコード、データおよびスタックセグメントの例を以下に示します。

```
dseg    SEGMENT  word public 'data'
        .
        .
dseg    ENDS
cseg    SEGMENT word public 'code'
        .
        .
cseg    ENDS
sseg    SEGMENT word stack 'stack'
        .
        .
sseg    ENDS
```

正しくネストされたセグメントの例を次に示します

```
cseg    SEGMENT
        .
        .
dseg    SEGMENT
        .
        .
dseg    ENDS
        .
        .
cseg    ENDS
```

CGA 表示のスクリーンメモリにオーバーレイできるセグメントを以下に示します。

```
scrseg  SEGMENT at 0B800h
        .
        . ;Template for screen memory goes here.
scrseg  ENDS
```

最後に16ビットおよび32ビットの80386 セグメントの例を以下に示します。

```
dseg      SEGMENT dword public use32 'data'
          .
          .
dseg      ENDS
cseg16    SEGMENT word public use16 'code16'
          .
          .
cseg16    END
cseg16    SEGMENT dword public use32 'code32'

          .
cseg32    ENDS
```

参照： SEGMENT(3.7 節)、ENDS(6.6.2 項)

3.9 GROUPディレクティブ

形式： groupname GROUP segname,...

GROUP ディレクティブは、1つ以上のセグメントのグループを定義するのに使われます。これにより、グループ中のすべてのセグメントはそのグループの最初のセグメントの持つベースアドレスに相対でリロケートされます。groupname パラメータは、グループ名を定義します。また、group パラメータは一意でなければなりません。segname パラメータは、モジュール中で定義されたセグメント名でなければなりません。

グループ定義は、メモリ中のセグメントの配置には影響しないことに注意してください。配置は、各セグメントのクラス指定子で行われます。グループディレクティブは、いくつかのセグメントを1つのセグメントレジスタでアクセスし、同じセグメントベースに相対でリロケートするというためにだけ使います。グループに属さないセグメントをグループに属するセグメントの間に配置させることは可能です。ただし、グループ内の最後のセグメントの終りが、最初のセグメントの始まりから4ギガバイト（80386 リンクの場合）または64Kバイト（その他のターゲットCPU でリンクした場合）を超えてはならないという制約があります。これは、80386 リンクでは決して発生しないが、8086リンクではフィクスアップオーバーフローエラーとして通知されます。

例：　Cプログラムのほとんどは、１グループに組み合されているデータ用セグメントをいくつか持ちます。定数データ用、変数用、スタック用のセグメントがあります。しかし、実行時、この３つのセグメントのアクセスはすべてDSセグメントレジスタから行ないます。次のコードは386｜ASM にこのプログラム構造を知らせるのに使われます。これにより、リンカに対する正しいリロケートの情報が生成できます。

```
const     SEGMENT word public 'CONST'
          .
          .
const     ENDS
data      SEGMENT word public 'DATA'
          .
          .
data      ENDS
stack     SEGMENT word stack 'STACK'
          .
          .
stack     ENDS

dgroup    GROUP const, data, stack
          ASSUME ds:dgroup
```

参照：　SEGMENT(3.7 節)、ASSUME(3.10節)、第11章

3.10　ASSUMEディレクティブ

形式：　ASSUME regname:segname regname:NOTHING,...
　　　　ASSUME NOTHING

ASSUMEディレクティブは、プログラム実行時にセグメントレジスタがとる値をアセンブラに知らせます。regname パラメータは、セグメントレジスタ：CS、DS、ES、FS、GS、SSのうちのいずれかの名前でなければなりません（アセンブルのターゲットが80386 である場合にだけFSとGSを使用できる）。segname は、アセンブル中のモジュールで定義したセグメントまたはグループの名前でなければなりません。NOTHING というキーワードは、そのレジスタがプログラム中のいかなるセグメントをもポイントしないことを示すために、segname パラメータの代わりに使われます。またこのキーワードは、すべてのセグメントレジスタが、未定義値を持つことを簡単に示すためにregname パラメータの代わりにも使われます。１つのASSUMEディレクティブの中に同一レジスタ名を２回以上使用した場合、エラーは通知されず、最後の（右端の）ものが使われます。

アセンブラは、セグメントレジスタの想定を使ってデータおよび命令の参照のアドレス可能性を確立し、必要な所にセグメントオーバーライドバイトを生成します。セグメントレジスタの値の変更のたびに、その値をできるだけ正確にアセンブラに知らせることはプログラムにとっても利益になります。

例：

```
;   Assume CS points at the program's code and DS points
;   at nothing.
ASSUME cs:code, ds:nothing
;   No assumptions are valid.
ASSUME nothing

;  Assume CS points at the program's code and all other
;  registers point at the data group.

ASSUME cs:code, ds:dgroup, es:dgroup, ss:dgroup
ASSUME fs:dgroup, gs:dgroup
```

参照： SEGMENT(3.7 節)、GROUP(3.9 節)、第11章

3.11 ロケーションカウンタ制御ディレクティブ

アセンブラは、モジュール中で定義されたセグメントごとのロケーションカウンタを維持します。ロケーションカウンタは次の命令またはデータバイトがセグメントにアセンブルされるオフセットです。386|ASM は必要であれば、ロケーションカウンタの現在値変更用のディレクティブをいくつか提供します。

3.11.1 ORG ディレクティブ

形式： ORG expression

ORG ディレクティブはロケーションカウンタの現在値を設定します。expressionパラメータは絶対アドレスまたはカレントセグメントのオフセットを求める式です。ORG ディレクティブは、現オープンセグメントのロケーションカウンタにだけ影響します。その他のセグメント（そのセグメントをネストするセグメントを含む）のロケーションカウンタは、影響されません。

ORG ディレクティブの処理後は、カレントセグメントの中のすべての命令またはデータバイトは、expressionで指定した位置に置かれます。

例：　　; Set the loc. ctr. to be offset 100h into the segment.

```
            ORG 100h.
        ; Create an array which can be accessed two different ways.
        ;  depending on which name is used.
            barray DB 100 dup(?)
            ORG barray
            warray DW 50 dup(?)

        ; Origin the locanion counter 10 bytes into barray.
          ORG barray+10
```

参照：　EVEN(3.11.2 項)、ALIGN(3.11.3 項)

3.11.2 EVENディレクティブ

形式：　EVEN

EVENディレクティブは、次に生成される命令またはデータバイトをワード境界に強制的に位置合わせします。ロケーションカウンタの現在値が奇数である場合、NOP 命令を挿入することによってその位置合わせを行います。偶数であれば、このディレクティブは何もしません。

EVENディレクティブは、BYTEアライン型をもつセグメントには使用できません。その理由は、386|LINKが他のセグメントとそれを組み合わせる時に、そのセグメントの持つベースアドレスが偶数か奇数かがアセンブラには分からないからです。

例：　　100 ワードの配列をワード境界に置く例を次に示します。

```
          EVEN
          warray DW 100 dup(?)
```

MOV 命令前にNOP バイトを強制的に生成する例を次に示します。

```
          cseg        SEGMENT use 16
                       ORG 100h
                       sub ax, 1000h
                       EVEN
                       mov bx, ax
           cseg        ENDS
```

参照：　ORG(3.11.1項)、ALIGN(3.11.3 項)

3.11.3 ALIGN ディレクティブ

形式：　ALIGN expression

ALIGN ディレクティブは、現オープンセグメントのロケーションカウンタをexpressionで指定したアラインメント境界に設定します。expressionは、２の累乗です。そうでないとエラーになります。

386|ASMは、指定されたアラインメント値がセグメントのアラインメント型にとって意味があるかどうかを確認しません。セグメントのロケーションカウンタをセグメントのアライン型の細分性より大きい値に合わせることは意味がありません。その理由は、セグメントがどのようにして 386|LINKによって他のセグメントと組み合わされるかをアセンブラは知らないからです。

ロケーションカウンタを指定された境界に設定するのに必要なバイト数が４バイト未満の場合、 386|ASM はNOP バイトを使ってスペースを埋めます。４バイト以上なら、386|ASMは、ロケーションカウンタを調整するだけです。これによって 386|LINKはスペースを、ゼロバイトで埋めます。この機能は、コードセグメント中の命令をワード境界またはダブルワード境界に置くのに役立ちます。

例：

```
;
; Align the location counter on a double word boundary
;
  ALIGN 4
;
; Align the location counter on a paragraph boundary.
;
  ALIGN 16
;
; Align the location counter on a page boundary.
;
  ALIGN 256
```

参照：　ORG(3.11.1 項)、EVEN(3.11.2項)

第4章 EQU および= ディレクティブを使った定数の定義

4.1 はじめに

アセンブルに使われる定数値は、シンボル名に置き換える方が望ましい。シンボル定数は、プログラムを読みやすくしたり、サポートしやすくします。386|ASM は、3種類のシンボル定数（整数値、アセンブラキーワードの別名および任意テキストの文字列）をサポートします。

シンボル定数は、EQU ディレクティブと等号（=）ディレクティブを使って定義します。これらのディレクティブは、変数とラベルの定義にも使用できます（詳しくは 5.2.3項と 6.5.3 項を参照）。

4.2 EQUディレクティブ

形式： name EQU expression

EQU ディレクティブは、3種類の定数（整数値、別名およびテキスト文字列）すべてを定義します。nameパラメータは、シンボル名を指定します。シンボル名は一意でなければなりません。アセンブラは、nameが出現するごとに、そのnameが整数でも別名でもテキスト文字列でも、その定数値に置換えられます。

EQU ディレクティブで定義したシンボルは 再定義できません。

4.2.1 EQUを使用した絶対定数の作成

形式： name EQU expression

アセンブラは、まずEQU ディレクティブの後のexpressionを整数式として値を求めます。それが絶対定数である場合、nameは、絶対定数シンボルとして、ユーザ定義シンボルテーブルに登録されます。絶対定数は、リロケート不能であり、データタイプも持ちません。そして、どんな未定義シンボルも後で定義されるシンボルも参照しません。その式がアドレス式（リロケータブル）の場合、nameは、ラベルか変数として、ユーザ定義シンボルテーブルに登録されます。詳細は、5.2.3 項、6.5.3 項を参照してください。

絶対定数シンボルは、整数が有効な場合ならどこでも、式の中で使用できます。その値は、PUBLICディレクティブを使えばプログラム中の他のモジュールでも有効にすることができます。

4.2.2　テキスト置換シンボルの作成

形式：　name EQU arbitrary text

EQU ディレクティブのあとに続く残り行をうまく整定数と認めることができない場合、アセンブラは代わりにそれをテキスト文字列としてセーブします。nameが出現するたびに、nameはEQC の後のarbitrary textに置換されます。これはパラメータなしのマクロを作成する簡単な方法です。

テキスト定数は、拡張のたびにアセンブラによって計算されることに注意してください。したがって、その定数値は、文字列中の1つのシンボル値に変更があれば、使用時ごとに変わります。一定の値が必要な場合、テキスト文字列の定義時、使用時は注意が必要です。

4.2.3　アセンブラキーワードの別名作成

形式：　name EQU keyword

アセンブラキーワードの別名は、実はテキストを置換する際の特別なケースです。EQU ディレクティブのあとに指定するテキストが単一のアセンブラキーワードの場合、nameはテキスト文字列ではなくそのkeyword のリファレンスとしてユーザ定義シンボルテーブルに登録されます。この置換形式は、シンボルテーブルのスペースが少なくすみ、そのテーブルの拡張も早くできます。

4.2.4　EQUの使用例

例：

```
;
; define some constants with EQU
;
```

```
c1         EQU 10h          ; absolute constant = 16
c2         EQU c1+10        ; absolute constant = 26
t1         EQU word ptr     ; t1 is a text string
arg1       EQU 8 [ebp ]     ; arg1 is a text string
t2         EQU c3+5         ; t2 is text because of
                            ; forward reference
c3         EQU 10h*10h      ; absolute constant = 256
stk_frm    EQU ebp          ; alias for keyword EBP

;
; use constants defined with EQU in instructions
;
mov        eax,c1           ; load 16 into EAX
mov        ebx,c2           ; load 26 into EBX
mov        ecx,arg1         ; load 8 [ebp ] into ECX
mov        edx,t2           ; load 261 into EDX
mov        esi,c3           ; load 256 into ESI
push       stk_frm          ; set up
mov        stk_frm esp      ; stack frame
```

参照：　=(4.3 節)、EQU(5.2.3項)、EQU(6.5.3項)、PUBLIC(7.2 節)、-DEFINE(2.2.7項)

4.3　＝（等号）ディレクティブ

形式：　name=expression

等号ディレクティブは、EQU ディレクティブとほとんど同じ方法で定数を作成します。しかし、制約があり、テキストの置換や別名シンボルは作成できません。

等号ディレクティブで定義した絶対定数は、整数が有効な場所ならどこでも、式の中で使用することができます。また、これらの値はPUBLICディレクティブを使って、他のモジュールにも有効にすることができます。

定数をEQU ディレクティブで定義した場合と等号ディレクティブで定義した場合との違いは、後者で定義した定数の再定義が、別の等号ディレクティブでその定数を使うことによっていつでもできることです。アセンブラは、参照時に、nameが出現するたびにnameをその時の最新の値に置換します。

例：

```
cl = 10h            ; absolute constant = 16
mov     ax,cl       ; load 16 into AX
cl = 20h            ; absolute constant = 32
mov     bx,cl       ; load 32 into BX
```

参照： EQU(4.2 節)、=(5.2.3項)、=(6.5.3項)、PUBLIC(7.2 節)

第5章 命令ラベル、制御トランスファ、およびプロシジャブロック

5.1 はじめに

80386 命令セット中の制御トランスファ命令の多くは、オペランドとして命令ラベルが必要です。命令ラベルは、アドレスとデータタイプをもつユーザ定義シンボルです。そのアドレスは、セグメントとオフセットの2つの属性を持ちます。セグメント属性は、ラベルの常駐するセグメントであり、オフセット属性は、そのラベルがセグメント中に置かれている位置です。ラベルのデータタイプは、NEARかFAR のいずれかです。このデータタイプはアセンブラが制御をそのラベルに移す方法を指定します。アセンブラは、この情報を使ってラベルへのCALLまたはJMPのための正しい命令コード、プロシジャ間のRETのための正しい命令コードを生成します。NEARラベルには、セグメント内トランスファで到達します。一方FAR ラベルには、セグメント間トランスファで到達します。制御トランスファ命令の中には、セグメント内トランスファだけをサポートするものがあります。それぞれの命令に有効なアドレスモードについては、付録Dを参照してください。8086プロセッサファミリのセグメントアーキテクチャが定めるNEAR制御トランスファおよびFAR 制御トランスファをよく知らない場合は、第11章を読んでから本章を読むことをすすめます。

命令ラベルとプロシジャ名には、PUBLICディレクティブとEXTRN ディレクティブを使って、グローバルにアクセスできます。

5.2 命令ラベル

ユーザ定義シンボルテーブル中に命令ラベルを作成する方法にはいくつかあります。それぞれの方法について以下に詳しく説明します。

5.2.1 ：（コロン）を使ったラベルの生成方法

形式： labelname:

最も簡単な命令ラベル作成方法は、ラベル名のあとにコロン（：）を続けることです。これで、labelname というラベルが生成されます。このラベルのデータタイプはNEARで、セグメント属性は現オープンセグメントです。そして、オフセット属性はロケーションカウンタのカレント値と等しくなります。ラベル名は、ソースライン上に最初に現れ、そして一意でなければなりません。

：（コロン）を使ってデータタイプFAR をもつラベルを作成する方法はありません。

例：　次の例は、アキュムレータに数値４があるかどうかによって、制御を２つの中の一方のNEARラベルに移す方法を示します。

```
cmp      eax,4
je       equal4
jmp      nequal4
equal4:

nequal4:
```

参照：　PUBLIC(7.2節)

5.2.2 LABELディレクティブを使ったラベルの作成

形式：　labelname LABEL type

LABEL ディレクティブは、ユーザ定義シンボルテーブル中にlabelname という名のラベルを作成します。typeパラメータは、ラベルのデータタイプを指定します。NEARかFAR のどちらかです。：（コロン）で作成したラベルのように、LABEL ディレクティブで作成したラベルのセグメント属性は現オープンセグメントであり、オフセット属性はロケーションカウンタのカレント値に等しくなります。

例：　２つのエントリポイントを同一ルーチンに定義する方法を次に示します。エントリポイントはそれぞれデータタイプNEARとFAR を持っています。

```
nearentry  LABEL near        ; Near entry point
farentry   LABEL far         ; Far entry point

           mov eax,5         ; Subroutine begins here
                .
                .
                .
```

参照： LABEL(6.5.2 項)、PUBLIC(7.2 節)

LABEL ディレクティブは、変数を作成できます。形式の説明は、6.5.2 項で行います。

5.2.3 EQUディレクティブと等号ディレクティブを使ったラベルの作成

形式： labelname EQU address expression
labelname = address expression

EQU ディレクティブまたは等号ディレクティブにのあとに続く式が、データタイプNEARかFAR を持つリロケータブルな数になるなら、 386｜ASM は、そのシンボルを定数ではなくラベルとしてシンボルテーブルに入力します。この方法を使って、任意のセグメント属性と任意のオフセット属性を持つラベルを作成できます。address espressionは、データタイプNEARまたはFAR をもつリロケータブルな数になる式です。新しいラベルは、式と同じセグメント属性と式の値と同じオフセット属性を持ちます。

例： EQU ディレクティブを使って命令ラベルを作成する例を以下に示します。

```
cseg      SEGMENT   word public 'code'
11        LABEL     far
csegl     ENDS

cseg2     SEGMENT   word public  'code'

labl      EQU       $          ; $ has a data type of NEAR
lab2      EQU       $+5        ; 5 bytes past the loc. ctr.
lab3:

; A far label at the same address as lab3
lab4      EQU far ptr lab3
; A far label 7 bytes past lab4
lab5      EQU lab4+7
; A near label at the current loc.ctr.
lab6      EQU THIS near
; A far label in segment csegl 4 bytes past 11
lab7      EQU 11+4
cseg2     ENDS
```

参照： EQU(4.2 節)、EQU(6.5.3項)、=(4.3節)、=(6.5.3項)、PUBLIC(7.2 節)、$(9.2.2 項)、THIS(9.3.13)

EQU ディレクティブと等号ディレクティブは、変数を作成することもできます。ディレクティブのあとに続くアドレス式が変数のデータタイプを持つ場合、変数が作成されます。詳細は6.5.3 項を参照してください。

5.2.4 アドレス式への制御トランスファ

制御トランスファ命令は通常、そのオペランドとして命令ラベルを必要とします。しかし、データタイプNEARまたはFAR をもつアドレス式を使うことはできます（NEARだけを使用できる命令もある）。したがって、宛先に置くラベルを作成することなく、任意のアドレス式で指定した位置に、制御をトランスファすることができます。

これがスタイルの理由でうまく行かない場合は、次式を使ったほうがよいこともあります。

例：

```
cseg     SEGMENT word public 'code'
labl     EQU     $
; a very simple address expression
         jmp     labl
; 5 bytes past labl
         jmp     labl+5
; a far jump to a near label
         jmp     far ptr nearlab
csegl    ENDS
cseg2    SEGMENT word public 'code'
nearlab  LABEL   near
cseg2    ENDS
```

5.2.5 間接形を使った制御トランスファ

CALL命令およびJMP 命令も変数をオペランドとして使用できます。アセンブラはオペランドとして変数をもつCALLまたはJMP 命令を見た場合、その命令の間接形を生成します。命令の実行後、制御は、変数の指している位置にトランスファされます。生成されたトランスファのデータタイプ（NEARまたはFAR)は、変数のデータタイプと、現オープンセグメントの使用属性によって異なります。たいてい、現オープンセグメントの使用属性と同じサイズの変数が、NEARトランスファを生成し、2バイト大きい変数が FARトランスファを生成します。このルールは曖昧なため、例外があります。詳しくは、付録Jを参照してください。

例：　8086ターゲットの命令をアセンブルしていると仮定した場合、次の命令は、間接制御トランスファを生成します：

```
label1    LABEL  near
nptr      DW       label1
fptr      DD       label1

call      nptr           ; Near transfer (16-bit ptr)
call      bx             ; Near transfer (16-bit reg)
call      fptr           ; Far transfer (32-bit ptr)
```

同様に、USE32 セグメント内で、80386 ターゲットの命令をアセンブルしているなら、

```
label2    LABEL near

nptr      DD       label2
fptr      DF       label2
call      nptr     ; Near transfer (32-bit ptr)
call      ebx      ; Near transfer (32-bit reg)
call      fptr     ; Far transfer (48-bit ptr)
```

5.3　プロシジャブロック

プロシジャブロックは、一つのプログラムを複数の論理的部分に分割するのに使われます。普通、プログラム中の各サブルーチンには、それぞれのプロシジャブロックが与えられます。プロシジャブロックにはサブルーチン名が与えられます。プロシジャブロック名はラベルと同じように複数の属性をもち、命令ラベルを使用できる場所ならどこでも、式中で使用できます。

プロシジャブロックの定義は、PROCディレクティブで開始し、ENDPディレクティブで終了します。一般に、プロシジャは、制御を呼出し元に戻すために、少なくともRET 命令を1つ持ちます。アセンブラは、プロシジャのデータタイプに対応した形式(NEARまたはFAR)のRET 命令を生成します。

プロシジャブロックはまた、任意数のローカル定義シンボルを持つことができます。ローカル定義シンボルは、定義されたプロシジャブロック中だけで参照されます。このため、別のプロシジャブロックにおいて、同じ名前をもつローカルシンボルが存在してもかまいません。

5.3.1 PROC ディレクティブ

形式：　procname PROC　{type}

PROCディレクティブは、プロシジャブロックの始まりをマークします。procnameパラメータは、プロシジャ名を指定します。この名前は一意でなければなりません。typeパラメータ（オプション）は、プロシジャタイプを指定します。このタイプは、NEARかFAR でなければなりません。プロシジャタイプが指定されていない場合、デフォルトのタイプNEARが使われます。プロシジャのアドレスはPUBLICディレクティブを使って他のモジュールにも有効にすることができます。

プロシジャ定義のネストはできますが、ネストは厳密にブロックストラクチャドでなければなりません。つまり、定義のオーバーラップはできません。

例：　AXレジスタに数字５をロードし、呼出し元に戻るプロシジャの例を次に説明します。

```
load5 PROC near

      mov ax,5
      ret

load5 ENDP
```

別プロシジャに正しくネストされたプロシジャの例を以下に示します。

```
outer PROC near      ; Start of outer procedure
  .
  .
inner PROC near      ; Start of nested procedure
  .
inner ENDP           ; End of nested procedure
  .
  .
outer ENDP           ; End of outer procedure
```

最後に、間違ってネストされたプロシジャの例を示します。

```
outer PROC near      ; Start of outer procedure


         .
```

```
inner PROC near      ; Start of nested procedure
  .
outer ENDP           ; ERROR - outer ends before
  .
inner ENDP           ; ERROR
```

参照： ENDP(5.3.2項)、PUBLIC(7.2 節)

5.3.2 ENDP ディレクティブ

形式： procname ENDP

ENDPディレクティブは、プロシジャブロックの定義を終了させます。procnameパラメータは、クローズしたいプロシジャ名を指定します。この名前は、最も近時にオープンしたプロシジャ名でなければなりません。procnameが最も近時にオープンしたプロシジャの名前でない場合、エラーが通知され、そののち最も近時にオープンしたプロシジャはクローズされます。

セグメントのクローズを現オープンプロシジャをクローズすることなく行うことはできます。しかし、その場合、ソースファイルの終了より前にセグメントの再オープンとプロシジャブロックのクローズが必要です。オープンしたのとは異なるセグメントでプロシジャをクローズすることは、誤りです。

参照： PROC(5.3.1項)

5.3.3 プロシジャブロック内のローカルシンボル定義

プロシジャブロック内でシンボル名の先頭文字に#を使うすべての変数、ラベルおよび定数の定義は、ローカルシンボル定義として処理されます。ローカルシンボル定義は、定義されたプロシジャ中だけで有効です。ただし、シンボル名は、そのプロシジャにとって一意でなければなりません。そのプロシジャをネストしているプロシジャの中では無効であり、また、定義されたプロジェクト内でネストされたプロシジャにおいても無効です。ローカルシンボル定義の形式は、通常のシンボル定義の形式とほぼ同じです。異なる点は、定義されたシンボル名の先頭は、#文字でなければならないということです。

ローカルシンボルは、それぞれに一意の名を付けずに一時ラベルと一時変数を生成できる便利な方法です。このシンボルは、条件付きジャンプ用、テンポラリ変数用、および実行時スタック中の関数パラメタのオフセット用の各ラベルとして使用すると便利です。

ローカルシンボルは、シンボルテーブル中に、通常のシンボルと同じ量のスペースをとることに注意してください。。その定義は、ネストしたプロシジャの終了後も、シンボルテーブル中に残っているが、有効でないだけです。

ローカルシンボルを、外部シンボルにすることはできません。また、PUBLICディレクティブでその値を他のモジュールに有効にすることもできません。

例：　プロシジャブロック内のローカルシンボルの使用例を以下に示します。この例は、ローカル定数、ラベル、および変数を作成し、使用します。

```
procl      PROC near

#argl  =   6

           mov    eax,#argl [esp]
           or     eax,eax
           jnz    #ll
           ret
#ll:
           mov    CS:#templ,eax
           ret

#templ     DD 0

procl      ENDP
```

参照：　ラベル(5.2節)、変数(6.5節)、定数（第４章)、シンボル形式(3.2.2項)

第6章 変数とデータ宣言

6.1 はじめに

データ宣言ディレクティブは、セグメント中にメモリブロックを作成し、初期化します（初期化はオプション）。386｜ASM は、アセンブラ定義のデータタイプを初期化するディレクティブを提供するほかに、プログラマによる任意のデータ構造の定義と初期化を許しています。アセンブラ定義データタイプは、DB、DW、DD、DF、DP、DQ、DTの各ディレクティブを使って初期化され、ユーザ定義データタイプはSTRUC ディレクティブとRECORDディレクティブを使って生成されます。PUBLICディレクティブおよびEXTRN ディレクティブを使用すると、グローバルにアクセスすることができます。

データ宣言ディレクティブは、変数も作成します。変数は、命令ラベルと同様、アドレスとデータタイプをもつユーザ定義シンボルです。そのアドレスは、セグメント属性とオフセットの２つから構成されています。セグメント属性は、変数が存在しているセグメントであり、オフセット属性は、そのセグメント中の変数の位置です。変数のデータタイプは、アセンブラ定義データタイプか、ユーザ定義の構造体名またはレコード名のいずれかです。

アセンブラは通常、命令の正しい形を選択するために命令オペランドのデータタイプを知っていなければなりません。オペランドの式に変数が使われている場合、そのオペランドにはその変数のデータタイプが与えられます。必要ならばデフォルトのタイプは PTR演算子を使えば変更できます。

6.2 データ宣言ディレクティブ

アセンブラ定義データタイプ用のデータ宣言ディレクティブを以下の項目で詳しく説明します。データ宣言ディレクティブの種類に関係なく、すべて同じ形式を使いますが、各ディレクティブは、その初期化リスト内の値が少し異なっています。

6.2.1 DBディレクティブ

形式： {name} DB value,...

DBディレクティブは、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメント中の１つ以上のバイトを初期化します。name（オプション）の指定がある場合、BYTEタイプの変数をユーザ定義シンボルテーブル中に作成します。この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値に等しくなります。

value パラメータは、整数、文字列定数、または未定義主記憶オペランド(?) になる式です。その値が整数の場合、１バイトを割当て、式の値に初期化します。値が未定義主記憶オペランドの場合１バイトを割り当てるが、初期値は与えられません。値が文字列定数の場合には、文字列中の文字ごとに、１バイトの割当てと初期化を行います。リロケータブルな値はDBディレクティブで使用できますが、 386|LINKがリロケータブルなリファレンスを分解し、それを１バイトにおさめようとすると、フィクスアップオーバーフローが発生することがあります。

ライン上にディレクティブのあとに続く値が複数ある場合、それらの値をコンマで区切ります。すると386|ASM は、その値を処理するため必要数のバイトの割当てと初期化を行います。

例：

```
factor  DB 10h
astring DB 'This is an ASCII string'
        DB 'A NULL terminated string',0
intexpr DB 12+4*4
listval DB 1,2,3,4,5
unknown DB ?
```

参照： DUP(6.3 節)、未定義主記憶オペランド(?)(6.4 節)、PUBLIC(7.2 節)、文字列定数(3.2節、3.4 節)

6.2.2 DW ディレクティブ

形式： {name} DW value,...

DWディレクティブは、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメントの中のワードを１つ以上（１ワードは２バイト）初期化します。name（オプション）の指定があると、WORDタイプの変数をユーザ定義シンボルテーブルに作成します。

この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値に等しくなります。

value パラメータは、整数、1文字または2文字の文字列定数、もしくは未定義主記憶オペランド(?) になる式です。その値が整数か1文字の文字列定数である場合、1ワードを割当て、式の値に初期化します。未定義主記憶オペランドの場合、1ワードは割当てるが、初期値は与えられません。DWディレクティブでリロケータブルな数は使用できますが、386 | LINKがリロケータブルなリファレンスを分解しようとすると、フィクスアップオーバーフローが発生することがあります。

ライン上にディレクティブのあとに続く値が複数ある場合、それらの値をコンマで区切ります。それぞれの値は、それぞれのメモリワードの割当てと初期化を行います。

例：

```
factor    DW 1000h
char1     DW  'A'
char2     DW  'AB'
          DW 12+4*4
listval   DW 1,2,3,4,5
relval    DW OFFSET factor
unknown   DW ?
```

参照： DUP(6.3 節)、未定義主記憶オペランド(?) (6.4節)、PUBLIC(7.2 節)

6.2.3 DD ディレクティブ

形式： {name} DD value,...

DDディレクティブは、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメント中のダブルワード1つ以上（1ダブルワードは4バイト）を初期化します。name（オプション）の指定がある場合、DWORD タイプの変数をユーザ定義シンボルテーブルに作成します。この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値に等しくなります。

value パラメータは、整数、1文字長から4文字長の文字列定数、4バイトの実数、4バイトのコード化された実数または未定義主記憶オペランド(?) になる式です。その値が整数か1文字の文字列定数の場合、ダブルワード1つを割当て、式の値に初期化します。未定義主記憶オペランドの場合、1ダブルワードを割当てるますが、初期値は与えられません。実数の場合には、4バイトを割当て、その数値は80287 形式の短実数として記憶されます。

リロケータブルなvalue の場合、386|ASM は、アセンブルのターゲットCPU の種類によって、別々のオブジェクトコードを出力します。80386 なら、リロケータブルな式は、リンカによって解析される32ビットのオフセットとして出力されます。8086、80186または80286 の場合には、リロケータブルな値は、16ビットセグメントセレクタを持つ16ビットオフセットとして出力されます。これらの値は、リンカによって解析され、オフセットとセグメント部分になります。

ライン上にDDディレクティブのあとに続く値が複数ある場合、それらの値をコンマで区切ります。それぞれの値は、それぞれのダブルワードのメモリの割当てと初期化を行います。

例：
```
factor    DD 10000h
char1     DD 'A'
char2     DD 'ABCD'
          DD 10000h*10h+07Fh
encreal   DD 3F800000r ;1.0
realval   DD 1.0E-10
listval   DD 1,2,3.14159,4,5
reloc1    DD factor
reloc2    DD OFFSET factor
unknown   DD ?
```

参照： DUP(6.3 節)、未定義主記憶オペランド(?)(6.4 節)、PUBLIC(7.2節)

6.2.4 DF ディレクティブとDPディレクティブ

例：
```
{name} DF value,...
{name} DP value,...
```

これらのディレクティブ（同義であり、置換え可能）は、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメント中の48ビットのポインタワードを１つ以上初期化します。name（オプションの）指定があると、PWORD タイプの変数がユーザ定義シンボルテーブルに作成されます。この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値に等しくなります。

value パラメータは、６バイトの整数、１文字長から６文字長の文字列定数、リロケータブルな数または未定義主記憶オペランド(?) となります。その値が６バイトの整数または文字列定数の場合、６バイトを割当て、その指定数値に初期化します。未定義主記憶オペランドの場合、６バイトを割当てますが初期化はしません。

その値がリロケータブルな数の場合には、32ビットオフセットとして出力され、うしろに16ビットセグメントセレクタが続きます。これら値は、386|LINKによって解析され、再配置可能な式のオフセット部分と、そのあとに続くセグメント部分になります。

ライン上にこのディレクティブのあとに続く値が複数ある場合、それぞれの数値をコンマで区切ります。それぞれの数値は、それぞれの6バイトメモリの割当てと初期化を行います。

DFディレクティブおよびDPディレクティブは、アセンブルのターゲットCPU が80386 である場合にだけ使用できます。

例：
```
sixbyte   DP 123456789ABCh
char1     DP 'A'
          DP 'ABCDEF'
reloc1    DF sixbyte
listval   DF 1,2,char1,4,5,sixexpr
unknown   DF ?
```

参照：　DUP(6.3 節)、未定義主記憶オペランド(?)(6.4 節)、PUBLIC(7.2 節)

6.2.5 DQ ディレクティブ

形式：　{name} DQ value,...

DQディレクティブは、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメント中のクワッドワードを（1クワッドワードは8バイト）1つ以上初期化します。name（オプション）の指定がある場合、QWORD タイプの変数がユーザ定義シンボルテーブルに作成されます。この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値と等しい。

value パラメタは、8バイトの整数、1文字長から8文字長の文字列定数、8バイトの実数、8バイトのコード化された実数、または未定義主記憶オペランド(?) となる。その値が整数または文字列定数である場合、クワッドワードを割当て、指定の値に初期化します。未定義主記憶オペランドの場合、クワッドワードを割当るが初期値は与えられません。実数の場合には、8バイトが割当てられその数値は、80287 形式のロング実数として記憶されます。

ライン上にディレクティブのあとに続く値が複数ある場合、それぞれの値をコンマで区切ります。それぞれの値は、それぞれのクワッドワードメモリの割当てと初期化を行います。

```
例：    quadint   DQ  123456789ABCDEF0h
        char1     DQ  'A'
        char2     DQ  'ABCDEFGH'
        realval   DQ  1.0E200
        encreal   DQ  3FF0000000000000r ;1.0
                  DQ  1,2,3.14159,4,5
        unknown   DQ  ?
```

参照： DUP(6.3 節),未定義主記憶オペランド(?)(6.4 節), PUBLIC(7.2 節)

6.2.6 DT ディレクティブ

形式： {name} DT value,...

DTディレクティブは、ライン上のディレクティブのあとに続く値のリストで、現オープンセグメント中の10バイト単位のメモリを1つ以上初期化します。name（オプション）の指定がある場合、TBYTE タイプの変数をユーザ定義シンボルテーブルに作成します。この変数のセグメント属性は現オープンセグメントであり、オフセットはロケーションカウンタのカレント値に等しくなります。

value パラメータは、10バイトの整数、1文字長から4文字長の文字列定数、10バイトの実数、10バイトのコード化された実数、10バイトのパック10進数、あるいは未定義な主記憶オペランド(?) となります。その値が整数か文字列定数の場合、10バイトを割当て、指定の数値に初期化します。未定義主記憶演算子の場合、10バイトを割当てますが初期化はしません。実数の場合、10バイトが割当てられその数値は、80287 形式の一時実数として記憶されます。最後に、カレントデフォルト基数が10であり、かつその数値の終りに基数指定子のない場合、その数値は10バイトの80287 パック10進数として記憶されます。
ライン上にディレクティブのあとに続く値が複数ある場合、それぞれの値はコンマで区切ります。それぞれの値は、それぞれ10バイト単位のメモリの割当てと初期化を行います。

DTディレクティブはデフォルト基数が10の時、全体が10進数字構成の定数を、パック10進数であるとみなすことに注意してください。DTディレクティブを使って、10バイトの整数値を定義したい場合は、10進数字構成の整数であることを示すために、その数字のあとに基数指定子"D" をつけます。

例：

```
        tbyteint  DT 123456789ABCDEF01234h
        char1     DT 'A'
                  DT 'ABCD'
        realval   DT 1.0E1000
        encreal   DT 3FFF8000000000000000r ;1.0
        packdec   DT 12345678901234567890
        listval   DT 1,2,3.14159, 4,5
        unknown   DT ?
```

参照： DUP(6.3 節)、未定義主記憶オペランド(?)(6.4 節)、PUBLIC(7.2 節)

6.3 DUP 演算子

形式： count DUP(value,....)

DUP 演算子は、与えられた初期値または値のリストを複数作成する特殊な目的の演算子です。count パラメータは、値リストの複製回数を指定します。このパラメータは、絶対整数値になる式であればかまいません。シンボルへのフォワードリファレンスは、この式の中では誤りです。

value パラメータ（またはパラメタリスト）は、括弧でくくります。このパラメータは任意の有効な式か、別のDUP 演算子にすることもできます。数値を複数指定した場合、リスト中の各エレメントは、コンマで区切ります。DUP 演算子のネストレベルは17まで許されます。

例： 複数個コピーされた値を初期化するためにデータ宣言ディレクティブを使ったDUP 演算子の使用方法を次に示します。

```
                  DB  500 DUP(?)
        length    equ 1000h
        warray    DW length  DUP(0)
        stack     DB 1024 DUP('STCK')
        alist     DD 100 DUP(1,2,3,4)
        nest1     DT 10 DUP(5 DUP(1.0))
        nest2     DB 5 DUP(1,2 DUP(?))
```

参照： 未定義主記憶オペランド(?)(6.4 節)

6.4 未定義主記憶オペランド(?)

未定義主記憶オペランド(?) は、ある場所がアセンブル時にもつ値が問題ないことをアセンブラに示します。そのため、初期値は指定されません。このオペランドは、データ宣言ディレクティブの値に使用できるが、式中で他の値との組合せはできません。

初期化しないで大量のメモリの割当てを行うために未定義主記憶オペランドをDUP 演算子と一緒に使うことがよくあります。

例：

```
DB ?               ; an undefined byte value
DD 1,2,?,4,5       ; an undefined double word in an array
DW 500 DUP(?)      ; an  array of 500 uninitialized words.
```

参照： DUP(6.3 節)

6.5 変数の作成

変数は、アドレス属性とデータタイプを持つユーザ定義シンボルです。変数にはスペースをとる必要はありません。したがって、２個の変数が同じメモリブロックを参照できます。本節では、アセンブラ定義データタイプを持つ変数の作成方法を説明します。ユーザ定義データタイプを持つ変数を作成する方法は、6.6 - 6.11節で説明します。またこれらの節ではストラクチャ宣言とレコード宣言についても説明します。

6.5.1 データ宣言ディレクティブを使った変数の作成

データ宣言ディレクティブは、ディレクティブの前に変数名を指定することによって変数を作成できます。この方法で、スペースの割当てを必要とする変数を簡単に作成できます。データ宣言ディレクティブで作成した変数のセグメント属性は、現オープンセグメントであり、オフセット属性はそのディレクティブが処理された時のロケーションカウンタの数値に等しくなります。

例： データ宣言ディレクティブを使った変数作成方法を以下に示します。この例は、ディレクティブのあとに続く値リストの単純値を使います。使用可能な値の詳しい説明は、それぞれのディレクティブの項で行います。

```
bytevar   DB ?             ; An undefined byte value.
wordvar   DW 14+15         ; A word with value 29.
dwordvar  DD 1.4           ; A dword variable with a
                           ; short real value.
tbytelst  DT 1,2,3,4,5     ; A list of 5 tbyte values.
bytearr   DB 1024 DUP(?)   ; A 1024  byte array.
```

参照： DUP(6.3 節)、未定義主記憶オペランド(?)(6.4 節)、PUBLIC(7.2 節)

6.5.2 LABELディレクティブを使った変数の作成

形式： name LABEL type

LABEL ディレクティブも、上述した形式を使って変数を作成します。nameパラメータは、作成する変数名であり、typeパラメータは、アセンブラ定義データタイプのいずれかです。LABEL ディレクティブで作成した変数のセグメント属性は、現オープンセグメントであり、オフセット属性はそのディレクティブが処理された時の、ロケーションカウンタの数値に等しくなります。

LABEL ディレクティブの処理時、ロケーションカウンタは変わらないので、このディレクティブで定義した変数は、スペースをとりません。 LABELディレクティブは、異なる方法で同じメモリブロックをアドレスする２つの変数を作成する便利な方法です。

例：

```
var1      LABEL word          ; A word variable.
var2      LABEL dword         ; A double word variable .

; An array which can be accessed as a byte or word
; array depending on which name is used.

barray   LABEL byte
warray   DW  512 DUP(?)
```

参照： LABEL(5.2.2 項)、PUBLIC(7.2 節)

6.5.3 EQUディレクティブと等号（＝）ディレクティブを使った変数の作成

形式： name EQU address expression
name = address expression

EQU ディレクティブまたは等号（＝）ディレクティブのあとに続く式が、アセンブラ定義データタイプの１つを持つリロケータブルな数になる場合、 386｜ASM は、シンボルテーブルにシンボルを定数ではなく変数として入れます。この方法を使うと、任意のセグメント属性とオフセット属性を持つ変数を作成できます。nameパラメータは変数の持つ名前であり、一意でなければなりません。address expressionは、リロケータブルな数になり、アセンブラ定義データタイプを持つ任意の式になります。この変数は、その式と同じセグメント属性を持ちます。

例： EQU ディレクティブと＝ディレクティブを使った変数作成方法の例を、次に示します。

```
var1      EQU      word ptr $          ; A word at the loc. ctr.
var2      EQU      byte ptr var1       ; A byte at the same
                                       ; address as var1.
foo       DB       5
var3      EQU      foo+5               ; A byte 5 past foo.
var4      =        THIS dword          ; A dword at the loc.ctr.
```

参照： EQU(4.2 節)、EQU(5.2.3項)、=(4.3節)、PUBLIC(7.2 節)、$(9.2.2項)、THIS(9.3.13 項)

EQU ディレクティブと等号（=）ディレクティブはまた、命令ラベルを作成します。ディレクティブのあとに続くアドレス式が、データタイプNEARまたはFAR を持つ場合、命令ラベルが作成されます。詳細は5.2.3 項を参照してください。

6.5.4 変数の参照

変数データタイプは、アセンブラが命令オペランドサイズ（バイト数）の決定と、互換性のないデータタイプ参照の検出を行うのに使用されます。

例：

```
wvar      LABEL    word
dvar      DD       0
          mov      wvar,1               ; move a word value
          mov      word PTR dvar,1      ; move a word value
          pop      dvar                 ; pop a double word
          mov      wvar,ax              ; valid statement
          mov      dword PTR wvar,eax   ; valid
          mov      wvar,eax             ; error
```

参照： PTR(9.3.7 項)

6.6 構造体定義

386｜ASM ではプログラマは任意のデータ構造の定義とそれら定義に対する名前の割当てを行うことができます。任意のデータ構造は１つ以上のフィールドの集合であり、それぞれはアセンブラ定義データタイプを持ちます。変数の作成時とデータの初期化時、アセンブラデータ宣言ディレクティブの代わりに、構造体名を使用することができます。このようにして、任意の構造体の変数の作成と初期化を行うことができます。構造体フィールドはまたそれに関連したデフォルト値を持つことができます。この値は、構造体の宣言時に使用されるので、宣言時には他の値は与えられません。

任意の構造体定義の作成プロシジャの説明を以降の項で行います。構造体定義はそれ自体何のメモリも割当てないことに注意してください。この定義は構造体に対するテンプレートの作成と、フィールドのデフォルト値のセーブを行うだけです。メモリを構造体に割当てるには、構造体宣言を使います。構造体宣言は、6.7 節で説明します。

6.6.1 STRUCディレクティブ

形式：　name STRUC

STRUC ディレクティブは、構造体定義の始まりをマークします。nameパラメータは作成する構造体名を指定します。これは一意でなければなりません。構造体定義はENDSディレクティブで終了し、STRUC ディレクティブとENDSディレクティブ間にあるすべての行は、構造体定義の一部として処理されます。構造体定義は、メモリを割当てないので、セグメントブロックの外側に置くこともできます。

6.6.2 ENDS ディレクティブ

形式：　name ENDS

ENDSディレクティブは、現構造体定義を終了させます。nameパラメータは、構造体定義を開始させるのに使われた名前と同じでなければなりません。ENDSディレクティブにより、アセンブラは現ストラクチャ定義を終了させてから、通常の文処理モードに戻ります。

ENDSディレクティブ（3.8 節参照）はまたオープンセグメントをクローズすることに注意してください。アセンブラは、構造体が現在定義されているかどうかを基に、どの形式のENDSを指定するかを決めます。

参照：　ENDS(3.8節)

6.6.3 構造体内でのフィールド定義

形式：
```
{fieldname} DB value,...
{fieldname} DW value,...
{fieldname} DD value,...
{fieldname} DP value,...
{fieldname} DF value,...
{fieldname} DQ value,...
{fieldname} DT value,...
```

いったん構造体定義を開始すると、ENDSディレクティブの前にあるすべての文はコメントかデータ宣言ディレクティブでなければなりません。構造体内のデータ宣言ディレクティブは構造体フィールド（オプション）を定義します。構造体フィールドは、そのディレクティブのデータタイプを持ち、サイズはディレクティブの割当てるメモリ量に等しくなります。ディレクティブのあとに続く値リストは、構造体の宣言時、そのフィールドのデフォルト値としてセーブされます。

一般的な構造体定義を次に示します。

```
name      STRUC
  ;    definitions for the structure fields go here.
name      ENDS
```

例：　平衡２進木中のノードになる構造体：

```
node      STRUC

key       DD 31 DUP(?)       ; The key for the node
balance   DB ?               ; The node's balance factor
lchild    DD ?               ; A pointer to the left child
rchild    DD ?               ; A pointer to the right child

node      ENDS
```

無名フィールドと初期値をもつ構造体：

```
pi        STRUC

sname     DB 'PI',0          ; Structure's ID
          DB 0               ; Pad byte for alignment
value     DQ 3.1415926       ; Initial Value

pi        ENDS
```

6.7　構造体宣言

上述したように、構造体定義は、データ用テンプレートだけを作成します。構造体にメモリを割当て初期化するには、構造体宣言を使います。この節は、１つまたは複数の構造体の割当てと初期化の方法、および、構造体変数の作成方法を説明します。

6.7.1 構造体名を使ったメモリの割当て

形式： {name} structurename < {value,... } >

いったん構造体を定義すると、その構造体名は、アセンブラ定義データタイプ用データ宣言ディレクティブと同じ方法で使用できます。この方法で構造体名を使用すると、アセンブラは、現オープンセグメント中の構造体にスペースを割当てることができます。

nameパラメータは、オプションであり、あれば変数を生成します。この変数は構造体のシンボルタイプを持ち、アドレス属性はロケーションカウンタのカレント値です。structurename パラメータは、STRUC ディレクティブを使って以前に定義された構造体名です。このstructurename パラメータのあとにカギ括弧< > をおかなければなりません。この括弧には構造体のデフォルトのフィールド値またはイニシャライザの代わりをする値のリスト（オプション）を入れます。初期化中の構造体内では１フィールドにつき１つの値が許可されます。値が複数の場合、コンマで区切ります。フィールドのイニシャライザを省略した場合、デフォルトのフィールド値が使われます。

6.7.2 イニシャライザ

386|ASM は、初期化できる構造体フィールドのタイプを制限します。単一の値または定数文字列である構造体フィールドだけが、イニシャライザにそのデフォルト値を変更させることができます。値のリストかDUP で割当てた複数値を持つフィールドは、構造体の割当て時、初期化することはできません。定数文字列の値は、その文字列長範囲内なら何バイトでも置換えることができます。イニシャライザが、置換えている文字列より小さい場合、フィールドの後のバイトは、スペース (20h)で埋められます。

6.7.3 DUP 演算子を使った多重構造体の作成

形式： {name} structurename count DUP (< {value,... } >)

DUP 演算子を使うと、同一の初期値セットを使用した複数の構造体の作成ができます。その宣言形式は、データ宣言ディレクティブの場合に使用したのと類似しています。

count パラメータは、いくつの構造体を作成するかを 386|ASM に知らせます。イニシャライザリスト（オプション）が、DUP 演算子のあとに続く括弧内に現われます。イニシャライザリストに使用できる値は、DUP 演算子の使用がされていない場合と同じです。

例： 構造体の定義と宣言の例をいくつか示します。

```
node    STRUC

key     DB 31 DUP(?)        ; Cannot be initialized
```

```
balance    DB 0                   ; Can be initialized
lchild     DD ?                   ; Can be initialized
rchild     DD ?                   ; Can be initialized

node       ENDS

; create a node using the structure's default values
node1 node<>

; create a node and set the balance factor to 0.
node2 node<,0>
; create a node and set the balance factor to 0 and
; the child pointers to two other nodes.
node3 node<,0,node1, node2>

; create 20 nodes with a balance factor of 0 and
; NULL child pointers.
nodearray node 20 DUP(<,0,0,0>)

pi       STRUC

name     DB  'PISTRUCT'        ; Can be initialized
         DB 0                  ; Can be initialized
value    DQ 3.1415926          ; Can be initialized
         DB 1,2,3,4            ; Cannot be initialized

pi       ENDS

; Create an instance using the default field values.
; Do not create a variable.
          pi <>

; Create an instance and override  the structure's name
; and value fields.
newpi     pi <'NEWPI',,3.1748>

; Create an array of 10 instances of the structure
; with no name and all fields zero except the last one
; which uses the default.
piarray pi 10 DUP(<'   ',0,0.0>)
```

参照： DUP(6.3節)、未定義主記憶オペランド(?)(6.4節)、STRUC(6.6.1項)

6.8 構造体の参照

構造体変数は、アドレス（セグメントとオフセット）属性を持つが、データタイプを持ちません。構造体中の個々のフィールドは、構造体フィールド演算子を（9.3.9 項参照）使うときわめて簡単に参照できます。式のデータタイプはフィールドのデータタイプから得られます。構造体フィールド演算子は、構造体中のフィールドオフセットをその構造体アドレスに加え、構造体フィールドの正しいアドレスを得ます。

例：
```
farp      STRUC
offs      DD        ?           ; offset in segment
segsel    DW        ?           ; segment selector
farp      ENDS
dseg      SEGMENT   word public
ptrarray  farp      10 DUP (< 0,dseg>)
dseg      ENDS

cseg      SEGMENT   word public

          mov       ax,ptrarray.segsel  ; load 1st ptr
          mov       es,ax               ; in array into
          mov       edi,ptrarray.offs   ; ES:EDI

          mov       ebx,OFFSET ptrarray ; load ptr to 2nd
          add       ebx,SIZE farp       ; struc in array
          mov       ax, [ebx].segsel    ; load 2nd ptr
          mov       fs,ax               ; in array into
          mov       esi, [ebx].offs     ; FS:ESI

cseg      ENDS
```

参照： SIZE(9.3.4項)、構造体フィールド演算子(9.3.9項)

6.9 RECORD ディレクティブ

形式： `name RECORD fname:width { = expr } ....`

RECORD ディレクティブを使うと、プログラマは８ビット、16ビット、または32ビットの値を名前付きの各サブフィールドに分散できます。それぞれは、任意のビット幅を持ちます。レコードの作成時にデフォルト値として使われる初期値を各ビットフィールドに与えることもできます。

nameパラメータは、定義中のレコードに名前を割当てます。この名前は一意でなければなりません。各フィールド定義は、独自のフィールド名とフィールド幅を持ち、これらはコロンで区切られています。デフォルトフィールド値（オプション）が指定されている場合、フィールド幅の直後に置き、前には等号を置きます。フィールド定義が複数ある場合はコンマで区切ります。

デフォルトフィールド値は、式によって指定されます。この式は指定されたフィールド幅に合った小さい定数値でなければなりません。デフォルト値の指定がない場合、フィールドにデフォルト値0が与えられます。

レコード内のフィールドは左から右へ割当てられ、先頭フィールドはディレクティブのあとに続くように、レコード中の左端ビットを使います。後続フィールドもレコードの残りのビットで左から右に割当てられます。

386|ASM は、次の規則でレコード定義を処理します。

1) レコード内の全フィールドの合計サイズ（単位：ビット）が計算され、そのレコードのサイズが指定されます。レコードのサイズは、8ビット、16ビットまたは32ビットです（32ビットのレコードは、ターゲットCPU が80386 である場合にだけ使用できる）。アセンブラは、全レコードがおさまるサイズの中で最小のものを指定します。
2) 次に、フィールド用のビットオフセットを指定します。上述したようにレコード中のフィールドは、RECORDディレクティブのあとに続いて、順に左から右に割当てられます。フィールドサイズの合計が、そのレコードを完全に埋めることができない場合、フィールドは右にシフトされます。これにより最終フィールドの最終のビットが、そのレコードのビットゼロの位置にきます。未使用の上位ビットは、ゼロにセットされます。

レコード定義自体はメモリを割当てないことに注意してください。この定義は、ただレコードに対するテンプレートの作成とフィールドのデフォルト値のセーブを行います。レコードにメモリを割当てるには、レコード宣言を使います。レコード宣言については6.10節で説明します。レコード定義はメモリを割当てないので、必要ならば、セグメントブロックの外側に置くことができます。

例：　RECORDディレクティブの例と、このディレクティブによって作成されるレコードフィールドのメモリ中での構成方法の例を、それぞれ以下に示します。フィールドのデフォルト値についてもメモリレイアウトの中に示します。

```
; An 8-bit record with two 4 bit fields.

rec1        RECORD field1:4, field2:4
```

Bit Number	7 ... 4	3 ... 0
	0	0
Field Name	Field 1	Field 2

```
; A 16-bit record which only uses 9 bits. The middle
; field has a default value of 3.

rec2        RECORD hibits :3, midbits:3=3, lobits:3
```

Bit Number	15 ... 8		7	... 0
	0	000	011	000
Field Name		hibits	midbits	lobits

```
; A 32-bit record which is made up of three one-byte
; fields having default values of A, B, and C.

rec3        RECORD  ch1 : 8='A', ch2 : 8='B', ch3 : 8='C'
```

Bit Number	31 ... 24	23 ... 16	15 ... 8	7 ... 0
	0	'A'	'B'	'C'
Field Name		ch1	ch2	ch3

6.10 レコード宣言

レコード定義もストラクチャ定義のように、データ用のテンプレートだけを作成します。レコードにメモリの割当てと初期化を行うには、レコード宣言を使います。この節では、1つまたは複数のレコードの割当てと初期化の方法およびレコード変数の作成方法を説明します。

6.10.1 レコード名を使ったメモリの割当て

形式： {name} recordname< {value,... } >

いったんレコードが定義されると、そのレコード名は、アセンブラ定義データタイプ用のデータ宣言ディレクティブと同じような方法で使用できます。この方法でレコード名を使うと、アセンブラは、現オープンセグメント中にレコードのためのスペースを割当てます。割当てるスペース量は、レコード中のフィールドの合計サイズに従って、１バイト、２バイトまたは４バイトになります。

nameパラメータは、オプションであり、あれば、変数を生成します。この変数のシンボルタイプはレコードで、アドレス属性はロケーションカウンタのカレント値です。さらに、レコード変数は、レコードサイズが１バイト、２バイトまたは４バイトかによって、BYTE, WORDまたはDWORD タイプの変数として参照されます。recordnameパラメータは、RECORDディレクティブで定義されたレコード名です。recordnameパラメータのあとにはカギ括弧< > を置きます。これにはレコードのデフォルトフィールド値の代わりに使われる値のリスト（オプション）を入れます。初期化中のレコード内に１フィールドにつき１つの値を使用できます。値が複数ある場合はコンマで区切ります。フィールドのイニシャライザを省略した場合は、定義時に与えられたデフォルト値が使われます。

レコードフィールドのイニシャライザは、絶対値でなければなりません。また初期化するフィールドの幅にも合わなければなりません。イニシャライザがリロケータブルか、そのフィールドには大きすぎる場合は、エラーが通知され、代わりにフィールドのデフォルト値が使われます。

6.10.2 DUP 演算子を使った複数レコードの作成

形式： {name} recordname count DUP(< {value,... } >)

DUP 演算子を使って、それぞれが同一の初期値セットを持つ複数のレコードを作成できます。この宣言の形式は、データ宣言ディレクティブで使われる形式と同じです。 countパラメータは、 386|ASM に、作成するレコードの数を知らせるので、イニシャライザリスト（オプション）がDUP 演算子のあとに続く括弧内に現われます。イニシャライザリストに使用できる値は、DUP 演算子を使用しなかった場合と同じです。

例：
```
recl        RECORD field1:4,field2:4

; a one-byte record using default values(in this
; case, zero since no defaults were given).
byte1  recl <>
```

```
; a one-byte record with the low nibble equal to 3.
byte2  recl <,3>

; a one-byte record with value 12h
byte4  recl <1,2>

; ten one-byte records with value 34h
bytearray recl 10 DUP (<3,4>)

rec3       RECORD ch1:8='A',ch2:8='B',ch3:8='C'

; a 32-bit record with value 00000043h
dword1 rec3<0,0>

; a 32-bit record with value 00123456h
dword2 rec3 <10h+2,4*12+4,56h>
```

参照： DUP(6.3 節)、未定義主記憶オペランド（?)(6.4節)、RECORD(6.9 節)

6.11 レコードの参照

レコード変数はアドレス（セグメントとオフセット）属性を持ちます。これらの変数は、そのサイズが8バイト、16バイト、または32ビットかによって BYTE、WORD、DWORD 変数として参照されます。WIDTH 演算子は、レコードかレコードフィールドビット幅を得るのに使われます。MASK演算子は、レコード中の定義されたビットまたは単一フィールドに対するビットマスクを得るのに使われます。SIZE演算子は、レコードのサイズをバイト（1、2、4、バイト）で示します。

例：

```
flags   RECORD   inuse:1,type:4
farray  flags    10 DUP (<0,0>)
flag2   PROC     near
        mov      al,farray+SIZE flags   ; load 2nd
                                        ; flag byte
        test     al,MASK inuse          ; call type
        jz       #not_used              ; check routine
        call     type_chk               ; if in use
#not_used:
        ret
flag2   ENDP
```

参照： WIDTH(9.3.5項)、MASK(9.3.5 項)、SIZE(9.3.4 項)

第7章 グローバルシンボル定義

7.1　はじめに

大きいプログラムをいくつかのモジュールに分割して、別々にアセンブルすることがしばしば必要になります。別々にアセンブルする時の問題は、プログラム全体でアクセス可能であったサブルーチンとデータが、アセンブル後は定義されているモジュール内でしかアクセス可能でないことです。グローバルシンボル管理ディレクティブは、変数、命令ラベル、および絶対定数を、１つのモジュールからプログラムの他のモジュールで参照できるようにすることによって、この問題を解決します。

PUBLICディレクティブは、アセンブル中のモジュール内のシンボル値を他のモジュールにも有効にするのに使われます。EXTRN ディレクティブは別のモジュール中で定義されたシンボル値を参照するのに使われます。これらのディレクティブの形式とオペレーションをさらに詳しく以下に説明します。

外部シンボルの参照に出会うと、アセンブラは式中でそのシンボルに対してゼロを与え、ファイル中のその位置でそのシンボルが参照された事実を記録します。モジュール用オブジェクトファイルの組合せ時、386 | LINKは、その式に、外部シンボルの実際の値を与えます。

7.2　PUBLIC ディレクティブ

形式：　PUBLIC name...

PUBLICディレクティブは、アセンブル中のモジュールで定義された１つまたは複数のシンボルの値を、他のモジュールにも使用できるようにします。nameパラメータは変数にも、命令ラベルにも、絶対定数にもなります。

シンボルをパブリックとして定義しても、その定義しているモジュールの中の使用方法には何ら影響しません。モジュールの中では、このシンボルはローカルシンボルと同じに扱われます。ただ１つ異なる点は、そのシンボルについての追加の情報が、アセンブラオブジェクト出力ファイルに記録され、この追加情報によって386 | LINKはシンボルに対する参照を解決することができる点です。

１つのグローバルシンボルには、PUBLIC定義が１つだけ存在しなければなりません。複数のモジュールで、あるシンボルがパブリックとして定義されると、386 | LINKはシンボル定義の重複を通知します。

例： 異なるシンボルの値を、他のモジュールに使用できるようにする方法を次に示します。

```
PUBLIC c1, c2, v1, v2, l1, l2, l3

c1 EQU 1000h
c2 = 'hi'
v1      DW 10FFh
v2      LABEL word

l1:
l2      LABEL near
l3      PROC far
```

参照： EXTRN(7.3 節)

7.3 EXTRNディレクティブ

形式： EXTRN name:type,...

EXTRN ディレクティブは、シンボルnameがアセンブル中のモジュールの外で定義され、typeというデータタイプを持つことをアセンブラに知らせます。この型指定子は、アセンブラ定義データタイプ、ストラクチャ定義名かレコード定義名、もしくはキーワードABSのいずれかです。ABS は、シンボルが絶対定数であることを示します。有効な型指定子のリストを次に示します。

BYTE
WORD
DWORD
PWORD/FWORD
QWORD

TBYTE
NEAR
FAR
ABS
structurename
recordname

シンボルの値はリンク時まではわかりませんが、アセンブラは、定義しているEXTRN 文の位置によって、変数とラベル用のセグメントを仮定します。EXTRN ディレクティブがセグメントブロック内部にある場合、それによって定義されるすべての変数とラベルは、そのセグメント内にあると仮定されます。アセンブラは必要であればオーバーライドバイトを生成します。ディレクティブがセグメントブロック中にない場合は、それによって定義されるラベルと変数は、特定のセグメント中にはないと仮定されます。この場合には、アセンブラは、それらを参照するのにオーバーライドバイトを生成する必要はないと判断します。いずれの場合も、式関連のセグメントは、必要ならば、セグメントオーバーライド演算子で指定することができます。

例：　現モジュールの外部変数、外部ラベル、外部定数を参照する方法を次に示します。

```
EXTRN number:WORD, scale:ABS, print_num:NEAR
foo     PROC near
        mov     ax,number       ; Get the number.
        add     ax,scale        ; Scale it.
        call    print_num       ; And print its value.
        ret                     ;
foo     ENDP
```

参照：　PUBLIC(7.2節)、セグメントオーバーライド(9.3.10 項)

7.4　外部シンボルをパブリックとして再定義する方法

386|ASM では、外部シンボルと定義されたシンボルをアセンブル中のモジュール内のパブリックシンボルとして再定義できますが、これには一つの制約があります。制約とは、それを外部シンボルと宣言したラインとその定義をパブリックに変えたラインとの間ではシンボルの参照はできないことです。すなわち、いったん外部シンボルをモジュール内で参照すると、その定義は外部シンボルとしてロックされ、変更できません。アセンブラエラーを避けるには、次の手順で外部シンボルをパブリックに正しく再定義しなければなりません。

```
    EXTRN foo:near      ; External Definnition.
      .
      .        ; foo cannot be referenced here.
      .
    PUBLIC foo          ; Redefintion as a public.
      .
      .        ; foo CAN be referenced here, and will follow
      .        ; the normal rules for forward references
      .        ; (the data type specified in the EXTRN is
      .        ; removed by the PUBLIC redefinition).
      .
   foo     PROC NEAR  ; Actual declaration of foo.
               ; foo can be referenced here.
```

シンボルのEXTRN 宣言、そのPUBLIC再定義、およびそれの参照というこの順序をくずすと、いずれもエラーとみなされます。

この機能を使うと、大規模システムのインターフェイス定義がより簡単になります。システムにおける全モジュール用のグローバル変数とエントリポイントは、全モジュールに含まれる一つのヘッダファイル中で外部シンボルとして定義されます。各モジュールには、この外部定義ファイルが含まれ、モジュール中に実際定義されているシンボルのためのPUBLIC定義が後に続きます。したがって、各モジュールは参照しようとする外部シンボルを宣言する必要はなく、すべてのモジュールは、システム中の全グローバルを定義するためのヘッダファイル１個を持ちます。

第8章 アセンブラ制御ディレクティブ

8.1 はじめに

アセンブラ制御ディレクティブは、全アセンブルに対してグローバルなパラメータを変更します。この章では、命令セット制御ディレクティブ、リスティングファイル制御ディレクティブ、条件付きディレクティブ、およびその他の制御ディレクティブを説明をします。

8.2 命令セットディレクティブ

命令セットディレクティブは、アセンブルのターゲットCPU と演算プロセッサを指定します。アセンブラディレクティブと命令ニューモニックによってはターゲットCPU に固有のものがありこれらのディレクティブによって使用可能／使用不能になります。ソースファイルの中に現れる命令セットディレクティブは、コマンドラインで指定された値をすべて無効にします。しかし、これらのディレクティブはモジュールの先頭セグメントの定義よりも前に指定することが必要です。

参照：　命令セットスイッチ（2.2.5 項）

8.2.1 .8086 ディレクティブ

形式：　.8086

.8086ディレクティブは、アセンブルのターゲットCPU を8086または8088に指定します。80186、80188、80286 または80386 に固有の命令はすべて無効となり、そのモジュール内で定義されたセグメントはすべて16ビットローケーションカウンタを使います。このディレクティブの処理後、386 | ASM は8086アセンブラになります。

8.2.2 .186 ディレクティブ

形式：　.186

.186ディレクティブは、アセンブルのターゲットCPU を80186 または80188 に指定します。80286 または80386 に固有の命令はすべて無効となり、そのモジュール内で定義されたセグメントはすべて16ビットローケーションカウンタを使います。

8.2.3 .286 ディレクティブと.286c ディレクティブ

形式：　.286
　　　　.286c

.286ディレクティブと.286c ディレクティブは同義であり、置き換えて使用できます。このディレクティブは、アセンブルのターゲットCPU として80286 を選択しますが、保護命令を有効にはしません。80386 に固有の命令はすべて無効となり、そのモジュールで定義されたセグメントはすべて16ビットローケーションカウンタを使います。

8.2.4 .286p ディレクティブ

形式：　.286p

.286p ディレクティブは、アセンブリ用ターゲットCPU を80286 に指定し、また保護命令を使用可能にします。80386 固有の命令はすべて無効となり、そのモジュールで定義されたセグメントはすべて16ビットローケーションカウンタを使います。

.286p ディレクティブは、うしろに.PROT ディレクティブがある.286ディレクティブと等価です。

8.2.5 .386 ディレクティブと.386c ディレクティブ

形式：　.386
　　　　.386c

.386ディレクティブと.386c ディレクティブは同義であり、置き換えて使用できます。これらのディレクティブは、アセンブリ用ターゲットCPU に80386 を指定しますが、保護命令を使用可能にはしません。ターゲットCPU が80386 の場合、セグメントのデフォルト使用タイプはUSE32 であり、32ビットローケーションカウンタが使われます。

80386 の非プロテクト命令セットは、386|ASM によって使用可能となるデフォルト命令セットです。ソースファイルに命令セットディレクティブがなく、コマンドライン上に命令セットスイッチがない場合、80386 非プロテクト命令セットが使用可能になります。

8.2.6 .386p ディレクティブ

形式： .386p

.386p ディレクティブは、アセンブルのターゲットCPU を80386 に指定し、保護命令を使用可能にします。セグメントのデフォルト使用タイプはUSE32 であり、32ビットローケーションカウンタが使われます。

.386p ディレクティブには、.PROT ディレクティブ（後述している）がある.386ディレクティブと等価です。

8.2.7 .PROT ディレクティブ

形式： .PROT

.PROT ディレクティブは、アセンブルのターゲットCPU を変更せずに保護命令を使用可能にします。ターゲットCPU が80286 か80386 の時だけこのディレクティブを使う意味がありますが、8086か80186 時に使っても、影響なく処理されます。

8.2.8 .8087 ディレクティブ

形式： .8087

.8087 ディレクティブは、8087浮動小数点コプロセッサ命令セットを使用可能にします。このディレクティブはアセンブルのターゲットCPU の変更はしません。

8.2.9 .287 ディレクティブ

形式： .287

.287ディレクティブは、80287 浮動小数点プロセッサ命令セットを使用可能にします。このディレクティブはアセンブルのターゲットCPU の変更はしません。8087と80287 との命令セットの違いは、80287 にはFSETPM命令があることです。また、FENI命令とFDISI命令は、ターゲットCPU が80287 であれば、オブジェクトコードを生成しません。

80287 が選択された時の重要な違いは、ターゲットCPU が80286 か80386 であり、かつターゲットコプロセッサが80287 の場合、 386|ASM は浮動小数点命令の前にFWAIT 命令を生成しないことです。その理由は、浮動小数点操作がプロセッサによって同期化され、FWAIT 命令が不要になるからです。

8.2.10 .387 ディレクティブ

形式：　.387

.387ディレクティブは、浮動小数点コプロセッサ命令セットを使用可能にします。このディレクティブはアセンブルのターゲットCPU を変更しません。リファレンス9で指定したすべての命令がサポートされます。

8.3 リストファイル制御ディレクティブ

リストファイル制御ディレクティブは、アセンブラリスティングファイルの形式を制御します。 386 | ASM は、ページタイトルとサブタイトルをセットするディレクティブとリスティングファイルにコピーされるソースラインを制御するディレクティブとを持ちます。

8.3.1 TITLE ディレクティブ

形式：　TITLE text

TITLE ディレクティブは、リスティングファイルページ見出しのタイトルフィールドをセットします。そのタイトルはライン上のディレクティブのあとに続くtextで示されます。1モジュールにつき1つのタイトルだけが使用できます。 386 | ASM が1モジュールに対し複数のタイトルを見つけると、エラーを通知し、最初のタイトルを使います。

モジュールにNAMEディレクティブで指定した名前がない場合は、モジュールタイトルの先頭6文字がその名前として使われます。

例：

```
        ;
        ; Set the title of the module to "386 | ASM  TITLE
        ; demonstration" and the name of the module
        ; to" 386 | AS".
        ;
      TITLE 386 | ASM TITLE demonstration
```

参照：　SUBTTL(8.3.2項)、name(3.3 節)

8.3.2 SUBTTL ディレクティブ

形式：　SUBTTL {text}

SUBTTLディレクティブは、リスティングファイルページ見出しのサブタイトルフィールドをセットします。そのタイトルはライン上のディレクティブのあとに続くtextで示されます。textパラメータはオプションです。テキストがディレクティブのあとに続かない場合、見出しのサブタイトルフィールドはクリヤされます。

アセンブル中のモジュールにSUBTTLディレクティブが複数出現してもかまいません。それぞれのディレクティブはリスティングファイルの次のページで有効になります。

例：
```
;
; Set the subtitle field to Subtitle #1.
;

SUBTTL Subtitle #1.
```

参照： TITLT(8.3.1 項)

8.3.3 PAGE ディレクティブ

形式：
```
PAGE {length} , {width}
PAGE+
PAGE
```

PAGEディレクティブは、使用する形式によって意味が異なります。length（オプション）およびwidth （オプション）、またはその一方が使用される場合は、リスティングファイルページの長さと幅を指定値に変えます。lengthパラメタは、10ラインから255 ラインの間でなければなりません。width パラメータは60文字から132 文字の間でなければなりません。長さまたは幅の指定がないと、その値は変更されません。

このディレクティブのうしろに正符号（+）を置くと，386 | ASM は、ページ見出しのページ番号の項目番号フィールドを増加させ、リスティングファイル中にページブレイクを作成します。

PAGEディレクティブを引数なしに使うと、 386 | ASM はリスティングファイル中にページブレイクを作成します。

例：
```
; Set the page length and width to 60 lines and
; 132 columns, respectively.

PAGE 60, 132
```

```
; Set the length to 60 without changing the width.
PAGE 60,

; Set the width without changing the length.
PAGE, 132

; Bumpe the section number of the number.
PAGE+

; Generate a page break in the listing file.
PAGE
```

8.3.4 .LIST ディレクティブ

形式：　.LIST

.LIST ディレクティブは、リスティングファイル中のプログラム文のリスティングを可能にします。このディレクティブは.XLISTディレクティブの効果を取り消すのに使われます。デフォルトでは、プログラム文のリスティングは、アセンブルが始まる時可能になります。

8.3.5 .XLISTディレクティブ

形式：　.XLIST

.XLISTディレクティブは、リスティングファイル中のプログラム文のリスティングを停止します。あとに続くすべてのソースラインは、.LIST ディレクティブが処理されるまでは、リスティングファイル中に表示されません。

8.3.6 .LISTI ディレクティブ

形式：　.LISTI

.LISTIディレクティブは、別のファイルからインクルードされたプログラム文を、リスティングファイル中にリストさせます。このディレクティブは、.XLISTI ディレクティブの効果を取り消すのに使われます。これは、アセンブリ開始時のデフォルト値です。

参照：　INCLUDE(3.5 節)

8.3.7 .XLISTI ディレクティブ

形式：　.XLISTI

.XLISTI ディレクティブは、別ファイルからインクルードされたプログラム文をリスティングファイルにリストするのを停止します。INCLUDE ディレクティブを使ってインクルードされた後続のソースラインは、.LISTIディレクティブが処理されるまではリスティングファイル中にリストされません。

参照：　INCLUDE(3.5節)

8.3.8 .LFCOND ディレクティブ

形式：　.LFCOND

.LFCOND ディレクティブは、偽の条件付きブロック中のテキストを、アセンブルされていなくても、アセンブラリスティングファイル中に表示させます。

参照：　条件付きアセンブリ(8.4節)

8.3.9 .SFCOND ディレクティブ

形式：　.SFCOND

.SFCOND ディレクティブは、それ以降のすべての偽の条件付きブロックをリスティングファイル中にリスティングすることを停止します。これはアセンブリの開始時のデフォルト値です。

参照：　条件付きアセンブリ(8.4節)

8.3.10 .TFCONDディレクティブ

形式：　.TFCOND

.TFCOND ディレクティブは、アセンブラの中の“偽の条件付きブロック”フラグの現在値を交互に変えます。偽の条件付きブロックフラグが現在リスティング中なら.TFCOND により停止されます。停止中の場合は、.TFCOND ディレクティブによってそのブロックは再びリストされます。

参照：　条件付きアセンブリ(8.4節)

8.3.11 .LALL ディレクティブ

形式： .LALL

.LALL ディレクティブは、マクロ展開によって作成されたすべてのプログラム文をリスティングファイル中にリストさせます。

参照： マクロ展開(10.3 節)

8.3.12 .SALL ディレクティブ

形式： .SALL

.SALL ディレクティブは、マクロ展開によって生成されたすべてのプログラム文の、リスティングファイルへのリスティングを停止します。

参照： マクロ展開(10.3 節)

8.3.13 .XALL ディレクティブ

形式： .XALL

.XALL ディレクティブは、オブジェクトコードを生成するマクロ展開からのプログラム文だけをリスティングファイルにリストさせます。これはアセンブル開始時のデフォルト値です。

参照： マクロ展開(10.3 節)

8.4 条件付きアセンブルディレクティブ

条件付きアセンブルディレクティブにより、プログラマは、あるブロックの命令が条件つきパラメータに基づいてアセンブルされるかどうかを指定できます。希望する条件の指定は、適切な条件付きアセンブルディレクティブを選択することによって行います。これについては、8.4.1 - 8.4.8 項で説明します。

条件付きアセンブルの最も簡単な形式は、条件付きアセンブルディレクティブで始まり、ENDIF ディレクティブによって終了するコードブロックです。条件付きブロック中の命令は、指定された条件に合致した場合にだけアセンブルさせます。この形式の条件付きアセンブルの一般的な構造を次に示します。

```
IF        (some condition is true)

.
.         (instructions to be assembled if the
.         condition is met)
.
ENDIF     (terminate the conditional block)
```

条件付きアセンブルディレクティブをELSEディレクティブと共に使うと、２つのコードブロックのうちアセンブルするほうを選択できます。条件付きブロックの"真の部分"といわれる第１のコードブロックは、条件付きアセンブルディレクティブで始まり、ELSEディレクティブで終了します。条件付きブロックの真の部分は、指定された条件に合致した場合にだけアセンブルされます。"偽の部分"とされる第２のコードブロックは、ELSEディレクティブで始まり、ENDIF ディレクティブで終了します。偽の部分は、指定された条件に合致しない場合にだけアセンブルされます。この形式の条件付きアセンブルの一般的構造を次に示します。

```
IF        (some condition is true)

.
.         (true portion: instructions to be
.         assembles if the condition is met)
.
ELSE

.
.         (false portion: instructions to be
          assembled if the condition is not met)
.
ENDIF      (terminate the conditional block)
```

条件付きブロックは、255 レベルまでネストできます。ENDIF は、前に置かれたIFまたはELSEのうち最も近いものと対をなしています。

8.4.1 IF ディレクティブ

形式： IF expression

IFディレクティブは、ライン上でこのディレクティブのあとに続くexpressionがゼロ以外の数値になる場合、条件付きブロックの真の部分にある命令をアセンブルします。

例：
```
IF      5
DB      0       ; will be assembled
ELSE
DB      1       ; will not be assembled
ENDIF
```

8.4.2 IFE ディレクティブ

形式： IFE expression

IFE ディレクティブは、ライン上でディレクティブのあとに続くexpressionがゼロになる場合、条件付きブロックの真の部分にある命令をアセンブルします。

例：
```
IFE     5
DB      0       ; will not be assembled
ENDIF
```

8.4.3 IFDEF ディレクティブ

形式： IFDEF name

IFDEF ディレクティブは、ライン上でディレクティブのあとに続くnameがユーザ定義シンボルまたはアセンブラキーワードである場合、条件付きブロックの真の部分にある命令をアセンブルします。nameパラメータは、前方参照であってはなりません。

例：
```
lab:
            IFDEF   lab
            jmp     lab         ; will be assembled
              ENDIF
```

参照： -DEFINE(2.2.7 項)

8.4.4 IFNDEF ディレクティブ

形式： IFNDEF name

IFNDEFディレクティブは、ライン上でディレクティブのあとに続くnameがユーザ定義シンボルまたはアセンブラキーワードでない場合、条件付きブロックの真の部分にある命令をアセンブルします。

例：

```
        IFNDEF  lab
lab     LABEL   near        ; define label only if not
                            ; already defined
        ENDIF
```

参照： DEFINE(2.2.7項)

8.4.5 IFB ディレクティブ

形式： IFB <string>

IFB ディレクティブは、カギ括弧< > 内のstringが空白の場合に、条件付きブロックの真の部分にある命令をアセンブルします。IFB はマクロパラメータの有無のテストに役立ちます。

例：

```
def_str     MACRO  param     ; define 0-terminated string
            IFB    <param>
            DB     0
            ELSE
            DB     '&param', 0
            ENDIF
            ENDM
```

参照： マクロ展開(10.3 節)、パラメータ置換(10.11節)

8.4.6 IFNB ディレクティブ

形式： IFNB <string>

IFNBディレクティブは、カギ括弧< > 内のstringが空白でない場合に、条件付きブロックの真の部分にある命令をアセンブルします。IFNBはマクロパラメタの有無のテストに役立ちます。

例： def_str

```
def_str     MACRO  param    ; define 0-terminated string
            IFNB   <param>
            DB     '&param'
            ENDIF
            DB     0
            ENDIF
```

参照： マクロ展開(10.3 節)、パラメータ置換(10.11節)

8.4.7 IFIDNディレクティブ

形式： IFIDN<str1>,<str2>

IFIDN ディレクティブは、カギ括弧< > 内の２つの string が同じ場合に、条件付きブロックの真の部分にある命令をアセンブルします。

例：

```
def_str     MACRO  flag    ; terminate string conditionally
            IFIDN  <flag>,<TRUE>
            DB     0
            ENDIF
            ENDM
```

参照： マクロ展開(10.3 節)、パラメータ置換(10.11節)

8.4.8 IFDIF ディレクティブ

形式： IFDIF <str1>,<str2>

IFDIF ディレクティブは、カギ括弧< > 内の２つのstringが異なる場合に、条件付きブロックの真の部分にある命令をアセンブルします。

例：

```
end_str     MACRO  flag    ; terminate string conditionally
            IFDIF  <flag>,<FALSE>
            DB     0
            ENDIF
            ENDM
```

参照： マクロ展開(10.3 節)、パラメータ置換(10.11節)

8.4.9 ELSE ディレクティブ

形式：　ELSE

ELSEディレクティブは、条件付きブロックの真の部分を終了させ、命令ブロックの始まりをマークします。これは真のブロックがアセンブルされない場合に、アセンブルされます。この命令ブロックを条件付きブロックの偽の部分と呼び、ENDIF ディレクティブによって終了させます。

8.4.10 ENDIF ディレクティブ

形式：　ENDIF

ENDIF ディレクティブは条件付きブロックを終了させます。このディレクティブは、ブロックの真の部分の最終文のあとまたは偽の部分（もしあれば）の最終文のあとに指定しなければなりません。

8.5 条件付きエラーディレクティブ

条件付きエラーディレクティブは、与えられた条件に一致する場合は、アセンブラエラーを強制的に発生させます。このディレクティブは、仮定が無効になった場合、エラーの通知に有用です。

8.5.1 .ERRディレクティブ

形式：　.ERR

.ERRディレクティブはアセンブラエラーを無条件に発生させます。

8.5.2 .ERRE ディレクティブ

形式：　.ERRE expression

.ERRE ディレクティブは、expressionがゼロの場合、アセンブラエラーを強制的に発生させます。

8.5.3 .ERRNZ ディレクティブ

形式：　.ERRNZ expression

.ERRNZディレクティブは、expressionがゼロ以外の場合に、アセンブラエラーを強制的に発生させます。

8.5.4 .ERRDEF ディレクティブ

形式： .ERRDEF name

.ERRDEF ディレクティブは、nameがユーザ定義シンボルまたはアセンブラキーワードの場合、アセンブラエラーを強制的に発生させます。

8.5.5 .ERRNDEF ディレクティブ

形式： .ERRNDEF name

.ERRNDEFディレクティブは、nameがユーザ定義シンボルまたはアセンブラキーワードでない場合に、アセンブラエラーを強制的に発生させます。

8.5.6 .ERRBディレクティブ

形式： .ERRB <string>

.ERRB ディレクティブは、stringが空白の場合に、アセンブラエラーを強制的に発生させます。

8.5.7 .ERRNB ディレクティブ

形式： .ERRNB <string>

.ERRNBディレクティブは、stringが空白でない場合に、アセンブラエラーを強制的に発生させます。

8.5.8 .ERRIDN ディレクティブ

形式： .ERRIDN <str1>,<str2>

.ERRIDN ディレクティブは、str1とstr2が同じである場合に、アセンブラエラーを強制的に発生させます。

8.5.9 .ERRDIF ディレクティブ

形式： .ERRDIF <str1>,<str2>

.ERRDIF ディレクティブは、文字列str1とstr2が異なる場合に、アセンブラエラーを強制的に発生させます。

8.6　その他の制御ディレクティブ

8.6.1　%OUTディレクティブ

形式：　%OUT text
　　　　%OUT1 text
　　　　%OUT2 text

%OUTディレクティブは、ライン上でディレクティブのあとに続くtextをユーザの端末にコピーします。%OUTディレクティブはアセンブルのパス１、２の両方でtextをコピーします。一方、%OUT1 と%OUT2 はそれぞれ、パス１またはパス２でtextをコピーします。

8.6.2　.RADIX ディレクティブ

形式：　.RADIX expression

.RADIXディレクティブは、ソースコード内の後続の数に対するデフォルト基数値をexpressionを使って設定します。expressionは常に基数10で計算され、カレントのデフォルト基数値は無視されます。expressionの値は２と16の間になければなりません。

第9章　式

9.1　はじめに

アセンブリ言語の命令またはディレクティブの多くはオペランドを必要とします。オペランドは定数でも、リロケータブルな値でも、80386 レジスタ名でもかまいません。

オペランドを必要とするほとんどの文では、単一の値を、定数またはリロケータブルな値になる複雑な式で置き換えることができます。式は、単一の値を計算する式演算子を使ってオペランドを組み合わせます。この章では、オペランドという語は、式中の単一定数値またはリロケータブル値を指します（単一の値か、単一の値になる式のいずれかを指すアセンブリ言語文のオペランドと対照的です）。

9.2　オペランド

式オペランドは、定数値、ユーザ定義シンボル（識別子）、アセンブラ予約語のうちのいずれかです。オペランドは、定数値かリロケータブルな値のいずれかです。リロケータブルな値とは、アセンブルされたオブジェクトコードのリンク時、またはリンクされたプログラムを実行のためメモリにロードする時に、変更（リロケート）される値です。

9.2.1　定数オペランド

定数オペランドは、数字、文字列定数、または定数値を持つユーザ定義シンボルです。定数値をとるユーザ定義シンボルのタイプと、それらのタイプが式中でとる値を表９－１に示します。

表９－１　定数シンボルタイプ

シンボルタイプ	値
整　定　数	シンボルを作成した時EQU ディレクティブまたは＝ディレクティブによって指定される値
構造体定義	構造体のサイズ（単位：バイト）
構造体フィールド	構造体内のフィールドのオフセット（単位：バイト）
レコード定義	レコードのサイズ（単位：ビット）
レコードフィールド	レコード内のフィールドのオフセット（単位：ビット）

参照：　定数（第４章）、構造体定義（6.6 節）、RECORD(6.9節)

9.2.2　リロケータブルオペランド

リロケータブルオペランドは、ユーザ定義シンボルであり、そのシンボルの最終値は、タスクイメージ作成のためにオブジェクトモジュールをリンクさせるまで、または、実行のためタスクイメージをメモリにロードするまでは、決定されません。表９－２はリロケータブルなユーザ定義シンボルとその値を示します。

表９－２　リロケータブルオペランド

シンボルタイプ	値
変数、構造体レコード、ラベルまたはプロシジャラベル	シンボルが定義されたセグメント内のシンボルのオフセット。最終値はリンク時に割当てられる。
$	カレントセグメント内のカレントオフセットを示す特殊シンボル。NEARラベルと同じ属性を持つ。最終値はリンク時に割当てられる。

シンボルタイプ	値
セグメントまたはグループ	8086、8088、80186、80188のプログラム、または80286、80386 のリアルモードプログラムの場合は、セグメントまたはグループをロードするメモリのパラグラフアドレス（各パラグラフは16バイト境界で始まる）である。 80386 のネイティブモードまたは80286 のプロテクトモードプログラムの場合には、オペレーティングシステムによって管理されるテーブル中のセグメントディスクリプタのインデックスである。セグメントディスクリプタは、メモリ中のセグメント位置と、そのセグメントについてのその他の情報を与える。いずれの場合も、最終値は、実行のためプログラムをメモリにロードする時に割当てられる。

参照： 変数（第６章）、SEGMENT(3.7 節),GROUP(3.9 節),第11章

9.2.3 アセンブラ予約語

アセンブラ予約語の中には有効な式オペランドもあります。有効なオペランドでない予約語が式中にあった時は、単に定数値ゼロを割当てられます。有効な式オペランドであるアセンブラ予約語と、それに割当てられる値を表９－３に示します。

表９－３ 予約語の値

予約語	値
レジスタ名	レジスタ名は、実効アドレス式（9.4 節）を命令コードと共に使う場合に有効である。
BYTE	定数値１
WORD	定数値２
DWORD	定数値４
PWORD	定数値６
QWORD	定数値８
TBYTE	定数値10
NEAR	定数値－１
FAR	定数値－２

9.2.4 前方参照

ソースコードファイル中に識別子を定義する前にその識別子（ユーザ定義シンボル）を使う式を、その識別子に対して前方参照であるといいます。前方参照が許されるのは、386 | ASM がたてた仮定によって、識別子についてアセンブルのパス1で生成されるオブジェクトコードが、オブジェクトファイルにパス2で書かれるのと少なくとも同じ長さをもつ場合です。

386 | ASMが違法な前方参照エラーを生成した場合、そのエラー状態は、識別子定義をそれを参照する文の前に移動させるか、前方参照した文中で識別子についてもっと多くの情報をアセンブラに与えることにより修正できます。PTR 演算子とセグメントオーバーライド演算子（：）は、前方参照された識別子についての追加情報をアセンブラに与えるのに使われます。

場合によっては、前方参照について十分な情報を386 | ASM に提供できないこともあります。そのときは、それを参照する文の前に定義を移動するしかありません。たとえば

```
long_str EQU 'This is a long string'
```

という文によって定義されるテキスト置換えシンボルは、どんな状況下でも、

```
DB long_str
```

という文によって前方参照することはできません。その理由は、アセンブラは文字列用スペースをパス1でどのくらい予約する必要があるかを知らなければならないからです。

参照： PTR(9.3.7 項)、セグメントオーバーライド(9.3.10 項)、SHORT(9.3.12 項)

9.3 演算子

式演算子は、複数オペランドを組み合わせて、単一定数値またはリロケータブルな値にします。演算子は、式の評価順序を決めるための優先順位のランキングを割当てられます。優先順位の高い演算子は、優先順位の低い演算子より先に評価されます。順位の等しい演算子は、左から右に評価されます。演算子の優先順位は、式中に括弧を使うことによって変更できます。括弧内の演算子は必ず隣接する演算子より先に評価されます。すべての式演算子の要約とその優先順位を表形式で付録Hに示します。

9.3.1 算術演算子

算術演算子は、標準的数学演算を行います。算術演算子とその形式およびその使用法を表9－4に示します。

表9－4　算術演算子

演算子	形　　式	使　　　　用
+	expr1 + expr2	2つのオペランドを加算する。両オペランドが定数の場合演算結果は定数である。片方のオペランドがリロケータブルであれば、結果はリロケータブルとなる。少なくとも片方のオペランドは定数でなければならない。
ー	expr1 - expr2	expr2をexpr1 から減算する。両オペランドが定数の場合、結果は定数である。両オペランドが同一セグメントに関してリロケータブルであれば、結果は定数である。左側オペランドがリロケータブルであり、右側オペランドが定数であれば、結果はリロケータブルとなる。リロケータブルオペランドの上記以外の使い方は許されない。
*	expr1 * expr2	2つのオペランドを積算する。両オペランドは定数でなければならない。結果は定数となる。
/	expr1 / expr2	expr1 をexpr2 で除算する。結果の小数部は切り捨てられる。両オペランドは定数でなければならない。結果は定数となる。
MOD	exprL MOD expr	モジュロ演算子。expr1 をexpr2 で割った時の剰余を与える。両オペランドは定数でなければならない。結果は定数となる。
単項+	+ expr	単項加算演算子。exprの値を求める。オペランドは定数でもリロケータブルな値でもよく、結果はオペランドと同じになる。
単項-	-　expr	単項減算演算子。exprの符号を逆にする。オペランドは定数でなければならない。結果は定数となる。

```
例：    var1    DB    0
        var2    DB    0
                DW    2 + 3            ; constant value 5
                DW    2 + var1         ; relocatable value
                DD    var2 + 3         ; relocatable value
                DB    2 - 3            ; constant value -1
                DW    var2 - var1      ; constant value 1
                DW    var2 - 1         ; relocatable value
                DD    2 * 3            ; constant value 6
                DD    9 / 2            ; constant value 4
                DD    9 MOD 2          ; constant value 1
                DB    +1               ; constant value 1
                DB    -1               ; constant value -1
                DB    +(2-3)           ; constant value -1
                DB    -(2-3)           ; constant value 1
```

9.3.2　ビット演算子

ビット演算子は数値をビット演算するために与えられています。ビット演算子の全オペランドは定数でなければなりません。結果は常に定数となります。ビット演算子を表９－５に示します。

表９－５　ビット演算子

演算子	形　　式	使　　用　　法
NOT	NOT expr	結果はexprの１の補数である。NOT 演算子が式中の最後の演算子である時、整数オーバーフローエラーを通知する場合の緩和ルールが適用される。すなわちフラッグワードまたはバイトの最上位ビットをオフにするために左記のようなNOT の使い方をする場合は、オーバーフローエラーを出さない。
AND	expr1 AND expr2	結果は２つのオペランドのビットAND(論理積）である。つまり、両オペランドが１であるビットはすべて結果も１となり、どちらかのオペランドがゼロであるビットはすべて結果はゼロとなる。

演算子	形　式	使　用　法
OR	expr1 OR expr2	結果は２つのオペランドのビット論理和（inclusive or）である。つまり、どちらかのオペランドが１であるビットはすべて結果は１、両オペランドがゼロであるビットはすべて結果はゼロとなる。
XOR	expr1 XOR expr2	結果は２つのオペランドのビット排他的論理和である。つまり、２つのオペランドが同一であるビットはすべて結果はゼロとなり、異なる場合は結果は１となる。
SHL	expr1 SHL expr2	結果は、expr2 の値で指定したビット数だけexpr1 を左にシフトする。ゼロビットは右にシフトされる。
SHR	expr1 SHR expr2	結果は、expr2 の値で指定したビット数だけexpr1 を右にシフトする。ゼロビットは左にシフトされる。
LOW	LOW expr	結果はexprのバイト０である。expr AND OFFh と等価である。
HIGH	HIGH expr	結果はexprのバイト１である。 (expr AND OFF00h) SHR 8 と等価である。
LOWW	LOWW expr	結果はexprのバイト０と１である。expr AND OFFFFh と等価である。
HIGHW	HIGHW expr	結果はexprのバイト２と３である。 (expr AND OFFFF0000h)SHR 16 と等価である。

例：

```
DB    NOT OFh               ;  OFOh
DB    OFFh AND OFh          ;  OFh
DB    OFOh OR ORh           ;  OFFh
DB    OFFh XOR OFh          ;  OFOh
DB    OFh SHR 2             ;  03ch
DB    OFFh SHR 3            ;  1Fh
DB    OFh SHR (1 + 3)       ;  OFOh
DD    LOW OFFA5A5FFh        ;  000000FFh
DD    HIGH OFFA5A5FFh       ;  000000A5h
DD    LOWW OFFA5A5FFh       ;  0000A5FFh
DD    HIGHW OFFA5A5FFh      ;  0000FFA5h
```

9.3.3　関係演算子

関係演算子は２つのオペランドを比較し、その結果TRUEまたはFALSE の値を返します。両オペランドは定数でなければなりません。TRUEの結果は、-1（すべてのビットを１にセット）を返し、FALSE の結果は、ゼロを返します。関係演算子の要約を表９－６に示します。

表９－６　関係演算子

演算子	形　　式	使　　用　　法
EQ	expr1 EQ expr2	両オペランドが等しい場合、結果はTRUEである。
NE	expr1 NE expr2	両オペランドが等しくない場合、結果はTRUEである。
LT	expr1 LT expr2	expr1 が expr2より小さい場合、結果はTRUEである。
GT	expr1 GT expr2	expr1 が expr2より大きい場合、結果はTRUEである。
LE	expr1 LE expr2	expr1 が expr2より小さいか等しい場合、結果はTRUEである。
GE	expr1 GE expr2	expr1 が expr2より大きいか等しい場合、結果はTRUEである。

例：

```
DW      2  EQ  3        ;  0
DW      2  NE  3        ;  0FFFFh
DB      2  LT  3        ;  0FFh
DW      2  GT  3        ;  0
DD      2  LE  3        ;  0FFFFFFFFh
DW      2  GE  3        ;  0
```

9.3.4　LENGTH 演算子とSIZE演算子

形式：　LENGTH variable
　　　　SIZE variable | structure | record

LENGTH演算子は変数に割当てられた主記憶のエレメント数を返します。各エレメントのサイズは、変数のデータタイプによって異なり、LENGTHによって返された値には関係しません。

SIZE演算子は変数に割当てられた主記憶のバイト数を返します。この数は(LENGTH variable) * (TYPE variable)と等しくなります。

SIZE演算子を、構造体変数とレコード変数と一緒に、そして構造体定義名およびレコード定義名と一緒に使って、構造体またはレコードのサイズ（単位：バイト）を知ることができます。LENGTH演算子は構造体またはレコードと一緒に使うことはできません。

例：

```
s1       STRUC
         DB      0
         DW      0
s1       ENDS
r1       RECORD  f1:8,f2:4
struc1   s1      <>
rec1     r1      <>

var1     DB      0
var2     DB      0,1,2
var3     DW      0,1,2
var4     DD      0,10, DUP (?), 1
         DW      LENGTH var1        ; 1
         DW      LENGTH var2        ; 3
         DW      LENGTH var3        ; 3
         DW      LENGTH var4        ; 12

         DW      SIZE   var1        ; 1
         DW      SIZE   var2        ; 3
         DW      SIZE   var3        ; 6
         DW      SIZE   var4        ; 48

         DW      SIZE   s1          ; 3
         DW      SIZE   struc1      ; 3
         DW      SIZE   r1          ; 2
         DW      SIZE   rec1        ; 2
```

参照：　変数（第６章）

9.3.5 WIDTH演算子とMASK演算子

形式：　WIDTH recdefname | recfldname
　　　　MASK recdefname | recfldname

WIDTH演算子は、レコード定義中の定義されたビット数またはレコードフィールドの中のビット数を返します（レコード定義のWIDTH は、レコード定義中に未定義ビットがある場合には、レコード定義時に割当てられた主記憶量と等しくないことに注意してください。レコードサイズは、常に８ビット、16ビット、32ビットであり、レコード名を直接使えば式演算子なしでも求めることができます）。

MASK演算子は、レコード定義中の定義されたビットのマスク、またはそのレコード中の単一フィールドのマスクを返します。

例：
```
rdef1   RECORD  rf1d1:2,rf1d2:8,rf1d3:1

rec1    rdef1   <1,5,0>          ; Allocate storage for a record

        DW      WIDTH rdef1      ; 11
        DW      WIDTH rf1d1      ; 2
        DW      WIDTH rf1d2      ; 8

        DW      MASK rdef1       ; 07FFh
        DW      MASK rf1d1       ; 0600h
        DW      MASK rf1d2       ; 01FEh
        DW      MASK rf1d3       ; 0001h

        DW      rdef1            ; 16 - size of record, in bits
        DW      rf1d1            ; 9  - offset of field, in bits
        DW      SIZE rdef1       ; 2  - size of record, in bytes
        DW      SIZE rec1        ; 2  - size of record, in bytes
```

参照：　RECORD(6.9節)

9.3.6 TYPE 演算子

形式：　TYPE expression

TYPE演算子は、expressionのデータタイプを示す定数値を返します。変数データタイプの場合、返す値は、データタイプのサイズ（単位：バイト）です。ラベルデータタイプの場合、返す値は、ラベルがNEARであればー１、FAR であれば-2である。すべての戻り値を表９－３に示します。

便宜上、TYPE演算子は、構造体とレコードについてはSIZE演算子と同義に使用できます。この演算子は、構造体またはレコードのサイズ（単位：バイト）を返します。

例：

```
var1        DB      0
var2        DD      0
lab1        LABEL   near
            DW      TYPE var1       ;1
            DW      TYPE var2       ;4
            DW      TYPE lab2       ;-1
```

参照： 変数（第６章）、ラベル（第５章）

9.3.7 PTR演算子

形式： type PTR expression

PTR 演算子は、expressionにtypeのデータタイプを与えます。ここでのtypeは、データタイプ予約語のうちの１つか、それらと等価の整数値（表９－３に示した）のうちの１つです。PTR 演算子は、前方参照されたシンボルのデータタイプについての情報をアセンブラに与える場合に有用です。これによってアセンブラは、前方参照されたシンボルがデフォルト値とは異なるデータタイプを持つという仮定をします。

例：

```
var1        DB      0
var2        DQ      0

            mov     al,byte PTR var2    ; load low byte of var2
            mov     ah,(TYPE var1)PTR  var2 + 1   ; load second
                                                  ; byte of var2
            call    far PTR lab1        ; tell assembler that
                                        ; foward referenced label
                                        ; is of type FAR
lab1        LABEL   far
```

参照： 前方参照(9.2.4項)

9.3.8 SEG演算子

形式： SEG expression

SEG 演算子は、expressionにセグメントセレクタ値を返します。その結果得た値は、16ビット幅のリロケータブルな値であり、プログラムを実行のためメモリにロードするまではその最終値は決定されません。

8086、8088、80186、80188のプログラムについては（または80286 、または80386 リアルモードプログラムについては）、セグメントセレクタはセグメントパラグラフアドレス（すなわち、セグメントが位置する16バイト境界のアドレス）です。セグメントパラグラフアドレスをバイトアドレスに変えるには、20ビットの幅のアドレス（単位：バイト）を作るために、そのアドレスを左に４ビットシフトします。

80286 プロテクトモードまたは386 ネィティブモードプログラムの場合、セグメントセレクタは、オペレーティングシステムによって維持されるテーブル中のセグメントディスクリプタのインデクスです。ディスクリプタテーブルの各欄は、セグメントが位置するアドレスとそのセグメント長およびセグメントについて他の情報を明示します。

例：
```
        mov     ax, SEG var1            ; load selector for segment variable
                                        ; var1 is located in
        mov     ax, SEG lab1            ; load selector for segment label
                                        ; lab1 is located in
```

参照：　第11章

9.3.9 構造体フィールド演算子

形式：　expression. strucfldname
　　　　expr1, expr2

構造体フィールドの演算子（. ）は、構造体中のフィールドをインデクスするのに使われます。この演算子は、構造体中の構造体フィールドのオフセットをexpressionに加え、expressionのデータタイプを構造体フィールドのデータタイプに変えます。

構造体フィールド演算子は、また、２つの式を加算する＋と同義に使用できます。

例：
```
        sdef1   STRUC
                sfld1   DB   0
                sfld2   DW   0
        sdef1   ENDS
        struc1  sdef1   <1,2>    ; allocate storage for a structure
                mov     al,struc1.sfld1      ; load byte at offset 0
                                             ; in struc1
                mov     bp,OFFSET struc1     ; load pointer to struc1
                mov     ax, [bp] .stfld2     ; load word at offset 1
                                             ; in struc1
```

参照：　構造体の参照(6.8節)

9.3.10　セグメントオーバーライド演算子

形式：　segregister: expression
　　　　segname: expression
　　　　groupname: expression

セグメントオーバーライド演算子（：）は、アセンブラが、デフォルトセグメントレジスタを使わないメモリ参照のためのオブジェクトコードを正しく作成する際の手助けをします。デフォルトでは、データの参照は、デフォルトセグメントレジスタとしてDS、SS、またはESを使います。どれを使うかは命令によって異なります（ほとんどの参照はDSを取り、スタック命令はSSを取ります。そしてESを取る文字列転送命令がいくつかあります）。

命令が、その命令に対して仮定したデフォルトセグメントレジスタによって指し示されないセグメントの中のデータを参照する場合、セグメントオーバーライドバイトが命令に先行します。このセグメントオーバーライドバイトは、データにアクセスするにはどのセグメントレジスタを使うかをプロセッサに知らせます。 386 | ASM は自動的に、シンボルを参照するラインより前に定義されているシンボルを参照するために、セグメントオーバーライドバイトを生成します。シンボルを前方参照する場合で、かつそのシンボルがデフォルトセグメントレジスタによって指し示されないセグメント中にある場合、セグメントオーバーライド演算子が、セグメントオーバーライドバイトが必要であることを386 | ASMに、知らせなければなりません。

セグメント名またはグループ名を、セグメントオーバーライドで使う場合、アセンブラが正しいオーバーライドバイトを生成できるように、どのセグメントレジスタがそのセグメントまたはグループを指すのかをアセンブラにあらかじめ知らせておく（ASSUMEディレクティブで）必要があります。

セグメントオーバーライドを変数が定義されたセグメント以外のセグメントにも使用できることに注意してください。たとえば、変数が定義されたセグメントを示すようにセグメントレジスタが設定されていない場合にはセグメントオーバーライドをこのように使う必要があることもあります。この場合、アセンブラは、必要とあればセグメントオーバーライドバイトを出力するばかりではなく、その変数のリロケータブルオフセットをセグメントオーバーライドに使われるセグメント（変数が定義されているセグメントではなく）に関して計算するように強制します。この計算は、プロセッサが正しい位置を参照するために必要である；セグメントレジスタが、セグメントオーバーライドに使われるセグメントを示すので、オフセットもそのセグメント相対でなければなりません。このタイプのセグメントオーバーライドは、80286 プロテクトモードまたは80386 の386 ネイティブモードプログラムでは使用できません。

例：

```
        ASSUME  cs:cseg, ds:dseg1, es:dseg2
dseg1   SEGMENT
var1    DW      0
dseg1   ENDS

dseg2   SEGMENT
var2    DW      0
dseg2   ENDS

cseg    SEGMENT

var3    DW      0

        mov     ax, var1          ; no segment override necessary
        mov     ax, var2          ; segment override to ES
                                  ; automatically generated by
                                  ; 386 | ASM
        mov     ax, var3          ; segment override to CS
                                  ; automatically generated by
                                  ; 386 | ASM
        mov     ax, forw1         ; no segment override necessary
        mov     ax, es:forw2      ; 386 | ASM must be told
                                  ; that a segment override
                                  ; is necessary
        mov     ax, dseg2:forw2   ; another way of doing the
                                  ; same thing
        mov     ax, cs:forw3      ; 386 | ASM must be told
                                  ; that a segment override
                                  ; is necessary
        mov     bp, OFFSET va2    ; load pointer to var2,
                                  ; relative to ES
        mov     ax, es: [bp]      ; 386 | ASM must be told
                                  ; that a segment override
                                  ; is necessary

forw3   DW      0

cseg    ENDS
```

```
dseg1   SEGMENT
forw1   DW        0
dseg1   ENDS

dseg2   SEGMENT
forw2   DW        0
dseg1   ENDS
```

参照： 第11章、前方参照(9.2.4項)、OFFSET(9.3.11項)、ASSUME(3.10節)

9.3.11 OFFSET演算子

形式： OFFSET expression

OFFSET演算子は、その演算子が定義されたセグメントに関する expression のオフセットを返します。この演算子は、ラベルまたは変数の実効アドレスを、レジスタにロードするのに役立ちます。

セグメントオーバーライドを式中で使う場合、返すオフセットは、変数またはラベルを定義しているセグメントではなく、オーバーライドに指定されたセグメント相対であることに注意してください。これはアセンブリ言語定義の欠点を補うのに必要です。変数をグループ内セグメント中で定義する場合で、かつそのグループがセグメントレジスタによって指される場合、変数のメモリ参照は、変数のオフセットをそのグループに関して正しく再配置します。しかし、OFFSET演算子は、グループ内ではなくセグメント内のその変数のオフセットを返すので、正しいオフセット値を得るためには、グループへのセグメントオーバーライドを使います。

OFFSET演算子をリロケート不能値上で使うと、その演算子は式の値を返します。構造体フィールドオフセットをイミディエイト値として使用しなければならない時には、正しいコードを得るためにこの機能を使用できます。

例：

```
        ASSUME  cs:cseg,ds:dgroup

dgroup  GROUP dseg1,dseg2

dseg1   SEGMENT
var1    DW        0
dseg1   ENDS
```

```
strl    STRUC
field1  DB          ?
field2  DW          ?
field3  DD          ?
strl    ENDS

dseg2   SEGMENT
var2    DW          0
dseg2   ENDS

cseg    SEGMENT

        mov         ax,var2             ; offset correctly relocated
        lea         bp,var2             ; offset correctly relocated
        mov         ax, [bp]

        mov         bp,OFFSET dgroup:var2   ; segmemt override
        mov         ax, [bp]                ; must be used to
                                            ; make offset be
                                            ; correctly relocated
        mov         ax,OFFSET labl          ; offset relocated correctly
        jmp         ax

labl:

        mov         ax,OFFSET field2        ; offset of structure
                                            ; field

cseg    ENDS
```

参照： SEGMENT(3.7 節)、GROUP(3.9 節)、セグメントオーバーライド(9.3.10 項)

9.3.12 SHORT 演算子

形式： SHORT label

SHORT 演算子は、ジャンプ命令のラベルに対して短い（8バイト）オフセットを生成するよう 386|ASM に命令します。このラベルは、ジャンプ命令から127 バイト以下になければなりません。これ以外はエラーになります。SHORT 演算子を上手に使うと、アセンブラは、コードの生成が少なくて済みます。

ただしラベルが前方参照されている場合には、SHORT 演算子を使う必要があります。上に定義したラベルのリファレンスについては、386 | ASM は、可能なら自動的に短いオフセットを生成します。

例：
```
        lab1:
                jmp         lab1          ; short offset automatically
                                          ; generated
                jmp         SHORT lab2    ; short offset
        lab2
```

9.3.13 THIS演算子

形式： THIS datatype

THIS演算子は、カレントセグメント中のカレントオフセットに等しいリロケータブルな値で、指定データタイプを持つオペランドを作成します。datatypeパラメータは、表9－3に記したアセンブラ予約語でなければなりません。

例：
```
                mov     al,THIS byte     ; load first byte of MOV opcod
        lab1    equ     THIS near        ; equivalent to lab1:or
                                         ; lab1 LABEL near
```

9.3.14 .TYPE 演算子

形式： .TYPE expression

.TYPE 演算子は、式についての情報を与える定数値を返します。表9－7は、返す値中で定義されるビットを表9－7に示します。返す値の中のその他のビットはすべて、ゼロであることが保証されます。

表9－7 .TYPE 演算子に対するビット定義

ビット	マスク	意　　味
0	01h	このビットが1であれば、式はラベルデータタイプを持つ。 それ以外はこのビットはゼロである。
1	02h	このビットが1であれば、式は変数データタイプを持つ。 それ以外はこのビットはゼロである。

ビット	マスク	意　　　　味
5	20h	このビットが１であれば、式中には未定義シンボルはない。式中に少なくとも未定義シンボルが１つあれば、返す値のこのビットを含むすべてのビットはゼロにセットされる。
7	80h	このビットが１であれば、式中には外部シンボルが定義されている。それ以外はこのビットはゼロである。

9.3.15 間接演算子

形式：　[effective address expression]
　　　　expr1 [expr2]

間接演算子［　］は実効アドレス式中のレジスタ間接であることを示します。これについては、9.4 節に詳しく説明します。この演算子は、２つの式を加算するために、＋演算子と同義に使用することもできます。

［　］中の演算は、（　）中の式と同じく、必ず隣接の演算より先に演算されることに注意してください。

例：

```
array   DB      20 DUP (?)
        mov     ax, [bp]          ; indirection through BP register
                ax,10 [array ]    ; load the 11th element in
                                  ; array
```

9.4 実効アドレスモード

8086ファミリプロセッサのアドレスモードは、ベースレジスタ、インデクスレジスタ、ディスプレースメント、およびインデクスレジスタにおけるスケールファクタの概念を使います。ベースレジスタやインデクスレジスタは、レジスタ間接アドレスモードで使われます。ディスプレースメントは８ビット16ビットまたは32ビットのイミディエイト値であり、この値はレジスタ間接アドレスモードで計算した実効アドレスに加えられます。ディスプレースメントは定数、再配置可能値のどちらでもかまいません。スケールファクタは、１、２、４、８であり、この値は、実効アドレスの計算に使う前に、インデクスレジスタの内容と掛け算をします。ベースレジスタまたはインデクスレジスタとして使用できるレジスタを表９－８に示します。80386でプログラマがアクセスできるすべてのレジスタの要約については、付録Ｆで説明します。

表９－８　8086ファミリにおけるベースレジスタおよびインデクスレジスタ

レジスタタイプ	8086, 8088 80186, 80286	80386
ベースレジスタ	BX, BP	BX, BP 。32ビットの任意の汎用レジスタ
インデクスレジスタ	SI, DI	SI, DI 。ESP 以外の32ビットの汎用レジスタ

9.4.1　イミディエイトオペランドモード

形式：　constant_expression

イミディエイトオペランドモードは、命令オペランドを命令の一部としてコード化するよう指定します。

例：

```
        var1    DW      0
                mov     ax,1                ; the value 1 is an
                                            ; immedeiate operand
                mov     ax,OFFSET var1      ; the offset value is
                                            ; an immediate operand
```

9.4.2　メモリ直接モード

形式：　relocatable_expression

メモリ直接モードでは、命令オペランドのアドレスを、命令の一部としてコード化します。

例：

```
        var1    DW      0
                mov     ax,var1         ; the operand is the data located
                                        ; at var1;  the address of var1 is
                                        ; given in  memory direct mode
```

9.4.3　レジスタ直接モード

形式：　register

レジスタ直接モードの場合、命令のオペランドは、レジスタの内容です。任意の汎用レジスタと、それにプラスするセグメントレジスタ、および80386 の制御レジスタ、テストレジスタ、ならびにデバッグレジスタが、レジスタ直接モードで使用できます。

例：

```
        mov     ax,bx       ; The operand is the contents of BX.
```

9.4.4 レジスタ間接モード

形式：
```
[register]
displacement [register]
displacement + [register]
[registerer +displacement ]
[registerer] . displacement]
[registerer] + displacement]
```

レジスタ間接モードの場合、命令オペランドのアドレスを得るために、レジスタ内容を、オプションのイミディエイトディスプレースメントに加えます。ベースレジスタまたはインデクスレジスタとして使用できるレジスタはどれも、このアドレスモードで使用できます。

例：
```
array   DB      20 DUP  (0)
        mov     ebx,OFFSET array
        mov     al, [ebx ]          ; register indirect
        mov     al, 10 [ebx ]       ; register indirect with
                                    ; displacement
        mov     al, [ebx+ 10]
        mov     al, [ebx ] .10

        mov     ebx,10
        mov     al,array [ebx ]     ; register indirect with
                                    ; register displacement
```

9.4.5 ベース付きインデクスモード

形式：
```
[basereg ] [indexreg]
displacement  [basereg ] [indexreg]
displacement + [basereg ]  [indexreg]
[basereg+displacement]  [indexreg]
[basereg ]  [indexreg + displacement ]
[basereg ]  [indexreg] . displacement
[basereg ]  [indexreg] + displacement
```

ベース付きインデクスモードの場合、命令オペランドの実効アドレスは、ベースレジスタおよびインデクスレジスタの内容と、オプションのイミディエイトディスプレースメントとの和により与えられます。

```
例：    array   DB      20 DUP  (0)
                mov     bp,OFFSET array
                mov     si,10
                mov     al, [bp]  [si]         ; based index addressing
                mov     al, [bp]  [si] .2      ; based index with
                                               ; displacement
```

9.4.6 スケール付きインデクスモード

```
形式：  [indexreg * constant_expression]
        displacement  [indexreg * constant_expression]
        displacement + [indexteg * constant_expression]
        [indexreg * constant_expression] . displacement
        [indexreg * constant_expression] + displacement
```

スケール付きインデクスモードの場合、命令オペランドのアドレスは、インデクスレジスタの内容と、スケールファクタ１、２、４、８のいずれかを掛け算し、次にこれを任意のイミディエイトディスプレースメントに加えることにより与えられます。このアドレスモードは、80386 だけに使用できます。

```
例：    array   DD      20 DUP  (0)
                mov     ecx,2                  ; 3rd entry in array
                mov     ecx,array  [ecx*4 ]    ; scaled index mode
```

9.4.7 ベース付きスケール付きインデクスモード

```
形式：  [basereg ]  [indexreg * constant_expression]
        displacement  [basereg ]  [indexreg * constant_expression]
        displacement + [basereg ]  [indexreg * constant_expression]
        [basereg + displacement]  [indexreg * constant_expression]
        [basereg ]  [indexreg * constant_expression] .  displacement
        [basereg ]  [indexreg * constant_expression] + displacement
```

ベース付きスケール付きインデクスモードの場合、命令オペランドのアドレスは、インデクスレジスタの内容にスケールファクタを掛け算し、次にこれをベースレジスタの内容とオプションのイミディエイトディスプレースメントに加えることにより与えられます。このアドレスモードは、80386 だけに使用できます。

```
例：    array   DD      20 DUP  (0)
                mov     ebx,offset array
                mov     ecx,2                       ; 3rd entry in array
                mov     eax, [edx ]  [ecx*4 ]       ; basesd scaled index mode
                mov     ax,2 [edx ]  [ecs*4 ]       ; load high byte of array
                                                    ; entry
```

第10章 マクロとリピートブロック

10.1 はじめに

マクロは、MACRO ディレクティブで作成するソースコード文の名前付きブロックです。いったんマクロを定義すると、そのマクロ名を参照することによってマクロの展開を行うことができます。マクロの展開時、マクロ定義の中のソースコード文のコピーがアセンブル中のソースコードに挿入されます。マクロはまた名前付きパラメータを持つこともあります。マクロ展開時、このパラメータの置換値が指定されます。

リピートブロックは、ソースコード文のシーケンスです。このシーケンスはリピートブロックディレクティブの制御下で指定回数反復されます。マクロと異なり、個別の定義や展開の概念はありません。ブロックは、単純に繰り返されます。

10.2 マクロ定義

形式：
```
name MACRO    {formalparam,... }
statements
ENDM
```

マクロ定義の作成は、MACRO ディレクティブとENDMディレクティブで、文のブロックをマクロ定義の中に囲むことによって行います。マクロにはname（必須）と形式パラメータ名のリスト（オプション）が与えられます。マクロの展開時、文ブロック中の形式パラメータ名が現われるたびに、そのパラメータは、マクロの呼出し時に指定した実パラメータ値に置換えられます。形式パラメータ名は、マクロ定義と共に保存されるので、ファイル中の別の場所で定義された他のシンボルとは衝突しません。マクロ名は、普通のユーザ定義シンボルであり、一意でなければなりません。マクロ定義内では、すべての形式パラメータ名は１ラインに表示しなければならないという点以外には、形式パラメータまたはソースコード文の数についての制限はありません。

マクロの定義時、オブジェクトコードは生成されません。しかし、マクロ定義はメモリの中にセーブされます。先へ行ってマクロをソースファイル中で展開する時に、オブジェクトコードが、生成されます。したがって、マクロ定義は、セグメントブロックの外側に置くこともできます。

二つのパラメータval1とval2および二つのソースコード文を持つalloc という名のマクロ定義を作成する例を次に示します。

```
alloc     MACRO     val1,val2
          DB        val1
          DB        val2
          ENDM
```

マクロは再定義できます。マクロのための新しい定義を見つけると、アセンブラは、古い定義を処分して新しい定義をセーブします。最初のマクロ定義と２回目のマクロ定義との間にマクロを展開する場合は、最初のマクロ定義を使います。２回目のマクロ定義のあとにマクロ展開する場合は２回目の定義を使います。

10.3 マクロ展開

形式： name {actualparam,... }

あらかじめ定義したマクロをnameで呼び出すと、マクロが展開されます。マクロ定義中のソース文は、ソースファイルの当該ポイントで挿入され、アセンブルされ、オブジェクトコードが生成されます。ソース文中に形式パラメータが現れるたびに、マクロ呼出し文からの対応する実パラメータによって置換えられます。実パラメータは、文字列として扱われ、コンマ、スペース、またはタブで区切られます。実パラメータのほうが、形式パラメータより少ない場合、足りない実パラメータには、null文字列が割当てられます。行中にコンマを２個入れることにより、実パラメータに対してnull文字列が渡されます。

マクロ展開時、リスティングファイル中に何をプリントするかについて次の三つのオプションがあります。(1) 挿入された文すべてをプリントする。(2) オブジェクトコードを生成させる文だけをプリントする。(3) 挿入文はどれもプリントしない。オプションの選択は、適切なリスティングファイルディレクティブ（8.3.11項－8.3.13項を参照）を使って行うことができます。デフォルトでは、オブジェクトコード生成文だけをプリントします。

alloc マクロ（10.2節で定義）と挿入されるソースコード文とを呼び出す二つの文を次に示します。

```
alloc       0, 1
```

は次のように展開します。

```
DB          0
DB          1
```

次の文

```
alloc       "that's" 'show biz'
```

は次のように展開します。

```
DB          "that's"
DB          'show biz'
```

マクロは、再帰的で、しかもネストすることができます。たとえば、次のペアのマクロ

定義：
```
term    MACRO
          DB        0 ; ; Terminate the string with a zero byte
          ENDM

bldstg  MACRO       val
        IFIDN       <val>,<5>
        term                            ;; terminate the string
        ELSE
        DB          'val&'              ;; next character in string
        bldstg      %val - 1            ;; continue loop
        ENDIF
        ENDM
```

は、次のように展開します。

```
bldstg      9
DB          '9'
DB          '8'
DB          '7'
DB          '6'
DB          0
```

参照：　.LALL(8.3.11項)、.SALL(8.3.12項)、条件付きアセンブル(8.4節)、パラメータ置換(10.11節)

10.4 REPTリピートブロック

形式：　REPT expression
　　　　statements
　　　　ENDM

REPTディレクティブは、REPTディレクティブとENDMディレクティブによって囲まれた文ブロックを、expression回数繰り返させます（expressionは定数でなければならない）。たとえば、次のリピートブロック

```
REPT    3
DB      0
ENDM
```

は、次のように展開します。

```
DB      0
DB      0
DB      0
```

10.5 IRPCリピートブロック

形式：　IRPC formalparam,string
　　　　statements
　　　　ENDM

IRPCディレクティブは、IRPCディレクティブとENDMディレクティブによって囲まれた文ブロックをstring中の文字数だけ繰り返させます。文ブロックを繰り返すごとに、string中の次の文字が、formalparam と置換えられます。たとえば、次のリピートブロック

```
IRPC    val,012
DB      val
ENDM
```

は、次のように展開します。

```
DB      0
```

```
DB        1
DB        2
```

参照： 条件付きアセンブル(8.4節)

10.6 IRPリピートブロック

形式： IRP formalparam,<actualparam,...>
statements
ENDM

IRP ディレクティブは、IRP ディレクテブとENDMディレクティブによって囲まれた文のブロックを、実パラメータリスト中の実パラメータごとに1 度繰り返させます。文ブロックを繰り返すたびに、現actualparam がformalparam と置換えられます。実パラメータリストの形式は、マクロ呼出しの実パラメータリストの場合とまったく同じです。実パラメータリストは、カギ括弧< > で囲まなければなりません。
たとえば、次のリピートブロック

```
IRP     val,<0, 'string', 1>
DB      val
ENDM
```

は、次のように展開します:

```
DB      0
DB      'string'
DB      1
```

参照： パラメータ置換(10.11節)、条件付きアセンブル(8.4節)

10.7 マクロとリピートブロックのコメント

形式： ; ; comment

マクロ定義またはリピートブロック定義内の２個のセミコロンで始まるコメントをマクロ／リピートブロックコメントと呼びます。マクロ／リピートブロックコメントは、マクロまたはリピートブロックの展開には現れません。これらのコメントは、コメントを表示することによって展開リストが乱雑になることなしに、マクロ定義またはリピートブロックを文書化します。たとえば、次のマクロ定義

```
define  MACRO   val1.val2
        DB      val1    ; this comment will show up
        DB      val2    ;; this one won't
        ENDM
```

は、次のように展開します。

```
define  0.1
DB      0           ; this comment will show up
DB      1
```

10.8 LOCALディレクティブ

形式： LOCAL formalpalam,...

LOCAL ディレクティブは、自動生成された形名??XXXX（XXXXは16進数）をマクロの展開ごとにformalparam に置換します。形式パラメータが複数の場合、コンマで区切らなければなりません。LOCAL ディレクティブは、マクロ定義ブロック内でだけ使用できます。リピートブロック内では使えません。

たとえば、次のマクロ定義が与えられた場合

```
chk-err   MACRO  limit
            LOCAL  skip
            cmp    ax,limit           ;; check value against
                                      ;; limit
            jle      skip             ;; skip call if OK
            call     error            ;; call error procedure
skip:
            ENDM
```

次のマクロ呼出し

```
chk-err 5
chk-err 10
```

は、次のコードを展開します。

```
            cmp         ax,5
            jle         ??0001
```

```
                call        error
??0000:
                cmp         ax,10
                jle         ??0001
                call        error
??0001:
```

10.9 EXITMディレクティブ

形式： EXITM

EXITM ディレクティブは、マクロまたはリピートブロックの展開を直ちに終了させます。たとえば、次のリピートブロック

```
IRPC        val,0123456789
IFIDN       <val>,<3>
EXITM
ELSE
DB          val
ENDIF
ENDM
```

は、次の文を生成します。

```
DB          0
DB          1
DB          2
```

参照： 条件付きアセンブル(8.4節)

10.10 PURGE ディレクティブ

形式： PURGE name,...

PURGE ディレクティブは、マクロまたはリピートブロックの展開に何の影響も与えません。既存の8086プログラムのPURGE 文によってエラーが生成されないようにするだけのために置かれます。

10.11 パラメータ置換

形式： &formalparam
formalparam&
&formalparam&

マクロの展開時、386｜ASMは、実パラメータをマクロ定義内の形式パラメータのそれぞれの代わりに置きます。形式パラメータ名（前後のシンボルとはスペース、タブまたは&文字で区切ってある）が発生するたびに、このパラメータ名は対応する実パラメータに取替えられます。"&" は、形式パラメータ名の一方または両側に置くことができます。この文字は、実パラメータが置かれる時に形式パラメータ名とともに除去されます。この文字は、また文字列定数中のパラメータの置換を強制的に行います。たとえば、次のリピートブロック

```
          IRP       val,<0 1 2>
var&val   DB        'This value is &val&'
          ENDM
```

は、次のように拡張します。

```
var0      DB        'This value is 0'
var1      DB        'This value is 1'
var2      DB        'This value is 2'
```

二つの形式パラメータを隣合わせに置く場合、両パラメータを強制的に置換するためそれらを２個の"&" 文字で区切ります。置換するたびに"&" 文字を１個取り去るのでこういうことが起こります。このルールは、形式パラメータがネストされたマクロ展開やリピートブロックの中の二つの別々のマクロ用のパラメータであっても適用されます。たとえば、ネストされたリピートブロックを持つ次のマクロ定義

```
define    MACRO     a,b
          DB        'a&&b'
          IRPC      c,012
a&&c      DB        b
          ENDM
          ENDM
```

は、次のように展開します。

```
        define     test 32
        DB         'test32'
test0   DB         32
test1   DB         32
test2   DB         32
```

参照： マクロ展開(10.3節)、IRPC(10.5節)、IRP(10.6節)、条件付きアセンブル(8.4節)

10.12 実パラメータリスト

マクロ展開に受け渡す実パラメータリストとIRP ディレクティブ用の実パラメータリストの形式は同じです。個々の実パラメータは、スペース、タブ、コンマ、セミコロン、またはカギ括弧< > で終了する文字列です。さらに、実パラメータリスト中でのみ有効な特別演算子がありますが、この演算子は任意の文字を実パラメータの一部にすることができます。

参照： マクロ展開(10.3 節)、IRP(10.6節)

10.12.1 リテラル文字演算子

形式： ! character

リテラル文字演算子(!) は、実パラメータリストの処理時、次文字をリテラル文字として処理させます。この演算子は、通常、特別な意味（パラメータセパレータまたはパラメータリスト演算子）を持つ文字をエスケープしてパラメータの一部にするために使用されます。

例：

```
define  MACRO     string
        DB        '&string'
        ENDM

        define    parameter! with! spaces
        DB        'parameter with spaces'

        define    a!;b; parameter with a ; in it
        DB        'a;b'

        define    0! !<! 1             ; escape < operator
        DB        '0 < 1'

        define    My! Gosh!!           ; escape ! operator
        DB        'My Gosh!'
```

10.12.2　リテラルテキスト演算子

形式：　<text>

リテラルテキスト演算子（＜＞）は、実パラメータの処理時カギ括弧< > 内のすべてのテキストをリテラルテキストとして処理させます。この演算子は、値をtextとする一つの実パラメータを作成します。そのライン上でこのパラメータより前にある実パラメータは、どれも＜文字で終ります。当該実パラメータは、＞文字で終了します。この演算子は、リテラル文字演算子と同様に、パラメータリスト中の区切り文字をエスケープすることができます。この演算子を、IRP ディレクティブ用の実パラメータリストを囲むのに必要なカギ括弧< > と混同しないでください（10.6節参照）。

カギ括弧< > 内でリテラルテキストとして取り扱われない文字は、他の二つのパラメータリスト演算子（！と％）と左右のカギ括弧です。！と％演算子は、カギ括弧< > 内でも有効であり、テキスト中の左右のカギ括弧は対になっていなければなりません（たとえば、<abc<de>fg> は文字列"abc<de>fg" を生成します）。テキストの中で対でないカギ括弧を必要とする場合、リテラル文字演算子を使用しなければなりません（たとえば、<abc!<defg> は文字列"abc<defg"を生成します）。

例：

```
define  MACRO   list
        DB      list
        ENDM
        define  <1,2,3>                 ; Escape comma separator
        DB      1,2,3

define  MACRO   str1,str2
        DB      '&str1&'
        DB      'str2&'
        ENDM

        define  this <is a string with spaces>
        DB      'this'
        DB      'is a string with spaces'

        define  this,< is a <string> with angle brackets>
        DB      'this'
        DB      'is a <string> with angle brackets'

        define  this < is a string with a !>>
        DB      'this'
        DB      'is a string with a >'
```

10.12.3　式演算子

形式：　%expression

式演算子は、expressionを評価して定数値を求め、その値をASCII 10進文字列に変換します。そして、実パラメータに値として結果の文字列を与えます。式として評価される文字列は、コンマ、セミコロン、感嘆符、左右のカギ括弧、または他のパーセント文字（すなわち、，；！＜＞%）で終了します。

通常、式演算子は、単一の実パラメータを返すためだけに使われます。式演算子は、実パラメータ中の最初の文字でなければなりません。テキスト文字列の中に%がある場合、通常、その他のテキスト文字として処理されます。例外は、カギ括弧< > で囲んだリテラルテキスト中ならどこでも%演算子として認識されることです。

例：

```
define  MACRO    val,string
        DB       val
        DB       '&string'
        ENDM

        define   %1 + 2,=3
        DB       3
        DB       '=3'

        define   %1 + 2 %3+4
        DB       3
        DB       '7'

        define   %3*4-2 <=10>
        DB       10
        DB       '=10'

        define   %3*4-2 ! !=! 5
        DB       10
        DB       '!= 5'

        define   %1 + 2,<is equal to %1 + 2,right?>
        DB       3
        DB       'is equal to 3,right?'
```

```
define    1,99%of100          ; notice % within string
                              ; has no effect
DB        1
DB        '99%of100'
```

10.12.4　文字列演算子

形式：　'string'
　　　　"string"

シングル引用符またはダブル引用符は、すべての文字（スペース、コンマ、その他のパラメータリスト演算子のような特殊文字を含む）をリテラルテキストとして処理するための実パラメータ中の文字列演算子として使用されます。文字列演算子がリテラルテキスト演算子と異なる点は次の三つです。(1) 文字列を区切る引用符は、実パラメータの一部として保存される。(2) すべての文字（他のパラメータリスト演算子を含む）は、文字列中でリテラルテキストとして処理される。(3) 文字列演算子は、実パラメータのテキスト中に組み込むことができる。つまり、文字列演算子は古い実パラメータを自動的に終了させ新しい実パラメータを始めることはありません。

例：

```
define  MACRO     str1,str2
        DB        str1
        DB        'str2&'
        ENDM

        define    "lot's! of <special>,chars%", 0
        DB        "lot's! of <special>,chars%"
        DB        '0'

        define    0, one"way to get"spaces
        DB        0
        DB        'one"way to get"spaces'
```

第11章 80386のプログラミング

この章では、8086プロセッサファミリのプログラミングメンバの特色について説明します。読者は、アセンブリ言語プログラミングと80386 プロセッサアーキテクチャについての基本的な知識は習得しているものとします。本章の目的は、アセンブリ言語プログラムの作成方法を既に知っている人が、80386 のプログラムを早く書けるように手助けすることです。本章では、アセンブリ言語のプログラム作成方法については解説しません。

11.1 リアルモードプログラミング

80386 はリアルモードと386 ネイティブドモードの二つの主な操作モードを持っています。リアルモードの場合、プロセッサは必然的に高速の8086プロセッサになります。現バージョンのMS-DOSはリアルモードでだけ動作するので、80386 プロセッサは、オペレーティングシステムがMS-DOSなら、通常、リアルモードで動作しています。本節では、80386または80286 （これもまたリアルモードとプロテクトモードを持つ）のリアルモードで実行するよう書かれたプログラム、および8086、8088、80186、80188プロセッサで実行するよう書かれたプログラムについて述べます。リアルモードで実行するためのHELLO.ASM というソースコードファイルの中のプログラムをアセンブルし、リンクするためには、次のように入力します。

```
386asm hello -8086
386link hello -8086
```

これで、ソースファイルHELLO.ASM をアセンブルしてオブジェクトファイルHELLO.OBJとリスティングファイルHELLO.LST を作成します。次に、そのオブジェクトファイルをリンクして、実行可能タスクイメージHELLO.EXE と、マップファイルHELLO.MAP を作成します。

8086ファミリアーキテクチャの重要な機能は、セグメンテーションの概念です。リアルモードにおいては、プログラムは、複数セグメントに分割されます。セグメントの最大長は64キロバイト（KB）です。プロセッサは、セグメントレジスタ（付録F参照）と呼ばれ

る特殊なレジスタセットを持っています。セグメント中のデータまたはコードをアクセスまたは実行する前に、そのセグメントを指定するようにセグメントレジスタをセットしなければなりません。

リアルモードにおいては、セグメントレジスタは、セグメントのパラグラフアドレスを持ちます。この場合、パラグラフは、アドレス空間内の16バイト境界です。セグメント内のフルアドレスは、そのセグメントパラグラフアドレスとセグメント内のオフセット(OKBから64KB)によって指定されます。プロセッサは、まずセグメントアドレスを左へ４ビットシフトしバイトアドレスに変え、最後にそれをセグメント内のオフセットに加えます。これによって、プロセッサは内部20ビットのアドレスを作成します。この内部アドレス作成方法は、リアルモードのプロセッサのアドレス空間を１メガバイト（内部アドレス幅が20ビット）に制限することに注意してください。理由は次節で述べますが、セグメントレジスタ中の値をセグメントセレクタと呼びます。そしてセグメントセレクタとセグメントオフセットからプロセッサによって作成された内部アドレスをリニアアドレスと呼びます。8086、8088、80186、80188 の場合、または80286 または80386 をリアルモードで実行する場合、リニアアドレスは常に、プロセッサがメモリにアクセスするためにアドレスバス上に発行する物理アドレスと同じです。

特定のセグメントレジスタは、プロセッサが特定の演算をすることを想定して考えます。CSレジスタは、カレントコードセグメントの指定用に（プロセッサによって）存在します。DSレジスタはカレントデータセグメントの指定用に、SSレジスタはカレントスタックセグメントの指定用に、ES、FSおよびGSの各セグメントは特別のデータセグメントの指定用にそれぞれ想定されています。たいていのプログラムは、実行するためにはCS、DS、SSを有効セグメントとして指定するようセットします。プログラムが特殊な使い方をしないかぎりES、FSまたはGSをセットする必要はありません。

プロセッサが行うすべてのメモリ参照については、内部メモリアドレスを計算するデフォルトのセグメントレジスタがあります。命令フェッチ操作に対しては、CSが使用されます。データ参照の場合は、DSが使用されます。スタック操作の場合（たとえば PUSH 、OP、およびEBP かESP レジスタを使った間接参照）、SSが使用されます。文字列操作のためには、ESだけが使用され、FSとGSは使用されません。どの操作をどのセグメントレジスタで行うかについての詳細は、付録Ｄとリファレンス６を参照してください。

セグメントオーバーライドバイトを使ってOPコードをプレフィクスすると、プロセッサがその命令のメモリ参照のために使うセグメントレジスタを変更することができます。これはメモリ参照がプロセッサ用のデフォルト値に従わない場合に386｜ASM が自動的に行います。たとえば、CSによって指定されたセグメント内の変数のデータ参照は、変数のアドレスを計算する上でDSレジスタではなくCSレジスタを使うようにプロセッサに伝えるセグメントオーバーライドバイトを持ちます。オーバーライドバイトを正しく生成するためには、386｜ASM はどのセグメントがどのセグメントレジスタによって指定されるかを知

る必要があります。これはASSUMEディレクティブ（3.10節参照）によってなされ、アセンブリ言語ソースコードファイルの必要な部分です。

MS-DOSがプログラムを実行のためにロードする時、CSセグメントレジスタは、実行開始のコードセグメントを指定するために、プログラムローダによって初期設定されます。DSレジスタは、プログラムについての情報ブロック（プログラムのデータセグメントではない）を指示するため、ローダによって初期設定されます。そのデータセグメントを指定するためのDSの初期設定は、通常、プログラムが実行開始時に最初に行う操作です。さらに、STACK 組み合わせ型（3.7.2 項参照）を持つすべてのセグメントは、リンカによって組み合わされて単一セグメントになります。ローダは、このセグメントを指定するためにSSを初期設定し、そしてセグメント内の最上位のオフセットを指定するためにSP（スタックポインタ）を初期設定します（8086ファミリでは、スタックは下方に伸びる）。プログラム中にSTACK 組合せ型を持つセグメントがない場合、SSとSPはローダで初期設定されません。すなわち、DSと同様にこれらのレジスタを初期設定するのはプログラムの責任です。

一般に、アプリケーションプログラムは、コードセグメント、データセグメントおよびスタックセグメントの三つのセグメントを持ちます。スタックセグメントとデータセグメントは、しばしば組み合わされてDSとSSの両方によって指定された単一セグメントになります。データセグメントはそのセグメントの下部に、スタックセグメントはセグメントの上部に置かれます。単純なプログラムにおいては、セグメントをただ一つ持つことも可能です。この場合、CS、DS、およびSSはすべて同じセグメントを指定します。

大きなプログラムでは、複数データセグメントか複数コードセグメント、またはその両方を必要とすることもあります（リアルモードにおける単一セグメントの長さは最大64KBであることに注意）。複数データセグメントの処理は、アクセスの必要のあるデータセグメントをどれでも指定するようにDSをセットするか、DSはメインデータセグメントを指定したままにして、その他のデータセグメントをアクセスするために特別のデータセグメントレジスタES、FS、およびGSを使うかのどちらかで行います。セグメントレジスタに新しい値を再ロードするたびに、正しいコードを生成できるように、ASSUMEディレクティブでその変更を386 | ASM に知らせねばならないことに注意してください。

複数コードセグメントの処理は、二つの形式のコール命令とジャンプ命令（NEARコールまたはNEARジャンプ、およびFAR コールまたはFAR ジャンプ）を持つことによって行ないます。NEAR制御トランスファは、CSレジスタによって現在指定されているセグメント内にあり、その命令のそのセグメント内のオフセットをオペランドとして持ちます。FAR 制御トランスファは、別のセグメント用であり、オペランドとしてセグメントセレクタ（リアルモードではセグメントパラグラフアドレス）と新しいセグメント内のオフセットを持ちます。FAR コールまたはFAR ジャンプが実行されると、プロセッサは、CSにその命令によって与えられた新しいセグメントセレクタをロードします。386 | ASM は、自動的に、正しい形式の呼出し命令またはジャンプ命令を生成します。386 | ASM は、コールまたはジ

ャンプの宛先であるラベルまたはプロシジャのデータタイプ（NEARまたはFAR ）から、どの形式を使うかを決定します（5.2 節と5.3.1 項参照）。リターン命令にも２形式があることに注意してください。386｜ASM は、FAR プロシジャ中にはFARリターン命令を、NEARプロシジャ中にはNEARリターン命令を生成することによって正しい形式の命令を生成します。念のために繰り返しますが、アセンブリ言語ソースファイル中の各々のコードセグメントの前には必ずASSUMEディレクティブを置いて、プログラム実行中にCSレジスタがそのセグメントを指定することを386 ｜ASM に伝えます。

CSレジスタ内容を正しく変更する方法はただ一つ、FAR コール命令またはFAR ジャンプ命令を使うか、CSレジスタを入れ替えるリターン命令を使うことである点に注意してください。CSレジスタを入れ替えるリターン命令は、RET 命令のFAR 形式と、割込み（IRET）命令からの復帰の二つです。

これらの概念を説明する簡単なプログラム例を以下に示します。この例では、プログラム実行時メッセージ"Hello world....."を端末に出力するために、MS-DOSシステムコールを行っています。このプログラムは三つのセグメント、すなわち、CSレジスタによって指定されるコードセグメント、DSレジスタによって指定されるデータセグメント、およびSSレジスタによって指定されるスタックセグメントを持ちます。このプログラムは、ES、FS、およびGSのレジスタを使わないので、それらを初期化する手間もかかりません。一見したところ、このプログラムは簡単そうなのでスタックセグメントも使わないように思えるかもしれません。実際には、INT（ソフトウェア割込み）命令が、フラッグワードと復帰アドレスをスタック上にプッシュします。通常のCPU 操作時のハードウエア割込みも、復帰アドレス用のスペースがスタック上に必要です。したがって、どのプログラムも、SSレジスタとSPレジスタをメモリ中の有効スタックを指定するようにセットします。

```
TITLE     hello.asm

;
; The .8086 directive can be used instead of the
; -8086 command line switch to cause programs to
; be assembled for execution in real mode on the
; 80386 or 80286, or for execution on the 8086,
; 8088, or 80186. The -8086 command line switch
; must still be used with the linker, however!
;
          .8086


;
; Tell the assembler which segments the CS, DS,
; and SS registers will point to
;
```

```
          ASSUME cs:cseg, ds:dseg, ss:sseg

;
; Align the data segment on a paragraph (16-byte)
; boundary and give it the standard combine type
; of PUBLIC.
;
data      SEGMENT PARA PUBLIC 'DATA'

;
; Message to be output, formatted as required by
; the MS-DOS system function call
;
hellomsg DB     'Hello world.....', 0Dh,0Ah,'$'

data      ENDS

;
; Allocate 1000 bytes in a segment with a
; combine type of STACK, which will cause the
; loader to initialize SS to point to it,
; and initialize SP to the highest offset.
;
sseg      SEGMENT PARA STACK 'STACK'

          DB      1000 DUP (?)

sseg      ENDS

;
; Also give the code segment the standard
; cmbine type of PUBLIC.
;
cseg      SEGMENT PARA PUBLIC 'CODE'

;
; The actual code to be executed is in
; a NEAR procedure.
;
print     PROC      NEAR
```

```
;
; Initialize DS to point to the data segment
;
            mov         ax,dseg
            mov         ds,ax

;
; Load the offset within the segment pointed
; to by DS of the message to be output. load
; the MS-DOS function code into All. and trap
; to the operating system to output the
; message to the terminal.
;
            mov         ah,09h
            mov         dx,OFFSET hellomsg
            int         21h

;
; Load the MS-DOS terminate process function
; code. and trap to the operating system
;
            mov         ah,4ch
            int         21h

print       ENDP

cseg        ENDS

;
; Identify the entry point of the program
; where the loader will transfer control after
; it loads the program and initializes the
; CS. SS. and SP registers.
;
          END       print
```

以上が全プログラムです。MS-DDSを実行する80386、8086、8088、80286 システム上で、本節の最初の例に示したように、そのファイル中にこのプログラムを持つファイルを作成し、そのファイルのアセンブルとリンクを行い、次に、これを実行して端末上にメッセージ"Hello world..."が表示されるのを確認できます。

セグメンテーション以外では、8086プロセッサファミリの機能で他のほとんどのプロセッサと共通でないものに、いくつかの命令による専用レジスタの使い方があります。ほとんどの命令にとって、汎用レジスタ（付録F参照）は相互に交換可能です。しかし、命令によっては、特定のレジスタにその命令のオペランドを置くことを想定するものがあります。たとえば、CXレジスタは、ある文字列命令のカウントオペランドを持ちます。AX、BX、DXのレジスタも、ある命令に対して特殊な使い方をします。あるレジスタを特殊な目的だけに使う命令を付録Dとリファレンス6で説明します。

11.2 386 ネイティブモードプログラミング

386 ネイティブモードは、80386 の全機能を使用できる操作モードです。このモードではかなり大量のアドレス空間へのアクセスと個々のセグメントの拡大（これによって、大きなプログラムの場合、複数セグメントを扱うむずかしさが取り除かれる）が可能になります。そして、算術演算用の32ビット幅オペランドのサポートと80386 の32ビット幅レジスタセットの使用が可能になります。80386 において386 ネイティブモード操作用のHELLO.ASM と呼ばれるファイルの簡単なプログラムのアセンブルとリンクを行うには、次のコマンドを入力します。

```
386asm hello
386link hello
```

アプリケーションプログラムからみると、80386 のリアルモードと386 ネイティブモードとの大きな違いは、セグメンテーションの処理方法です。386 ネイティブモードでは、セグメント内のオフセットの幅は、32ビットであり、単一セグメントの最大長は4ギガバイトです。セグメントの制御は、オペレーティングシステムによって維持される2種類のテーブル、つまりグローバルディスクリプタテーブル（GDT)とローカルディスクリプタテーブル（LDT)によって行います。グローバルディスクリプタテーブルはシステムにつき一つ、ローカルディスクリプタテーブルはプロセスごとに一つあります（したがって、マルチプロセスシステムではローカルディスクリプタテーブルは多数になる）。セグメントレジスタは、セグメントセレクタと呼ばれる値を持ち、これは、カレントプロセスに対してGDT かLDT かを選択するビットを持ちます。特権セグメントへのアクセスを制御するのに2ビットを使い、選択されたディスクリプタテーブル中のエントリをインデクスするのに13ビットを使います。したがって、GDT またはLDT には、最大8192のエントリを収容できます。各エントリのサイズは8バイトであり、各エントリはセグメントの開始アドレスとサイズを与えます。このエントリはまた、セグメントがコードセグメントかデータセグメントかを識別し、セグメントの特権レベルなどを識別するステータス情報も与えます。
386 ネイティブモードのセグメンテーション処理方法についての詳細はリファレンス6を参照してください。

386 ネイティブモードのリニアアドレスの作成は、次のように、プロセッサによってセグメントセレクタとセグメント内のオフセットから行います。幅16ビットのセグメントセレクタ値（セグメントレジスタ内）が、ディスクリプタテーブル中のエントリの指定に使われます。このエントリは、セグメント指定用に32ビット幅のリニアベースアドレスを与えます。次に、このセグメントベースアドレスが32ビット幅のセグメントオフセットに加えられ、32ビット幅のリニアアドレスが生成されます。プロセッサは、このプロセスをスピードアップするため、ディスクリプタテーブルエントリを内部キャシングしますが、これはプログラマには見えません。

80386 プロセッサはまた、チップ上でページングもサポートします。これが、セグメントセレクタとセグメントオフセットから形成されたアドレスを、物理アドレスではなくリニアアドレスと呼ぶ理由です。386 ネイティブモードではページングを無効にすることができ、この場合、リニアアドレスは、プロセッサがアドレスバス上に示す物理アドレスと同じです。ページングが有効の場合、チップ上のページングユニットは、32ビット幅のリニアアドレスを32ビット幅の物理アドレスに変換します。この変換はプログラマには見えず、無視してもかまいません。ページングの詳細は、リファレンス6を参照してください。

386 ネイティブモードセグメントは、コードセグメントかデータセグメントのどちらかを指定します。コードセグメントは書き込みできず、データセグメント中のプログラムは実行されません。したがって、386 ネイティブモードプログラムは少なくともコードセグメントとデータセグメントの二つのセグメントを持ちます。実際には、単一セグメントに対してディスクリプタテーブル中に二つのエントリを作成することによって単一セグメントを持つプログラムを持つことは可能です。一方のエントリがセグメントをコードセグメントとして識別し、他方のエントリがデータセグメントとして識別します。CSレジスタには、ディスクリプタテーブル中のコードセグメントエントリ用セレクタがロードされ、DSレジスタには、データセグメントエントリ用セレクタがロードされます。

386 ネイティブモードとリアルモードとの大きな違いをアプリケーションプログラマ側からみると、次のように要約できます。

1) セグメントレジスタの値は、リアルモードのようなアドレス値ではなく、ディスクリプタテーブル内でセグメントを記述するエントリへのインデクスになっています。
2) セグメント内のオフセット幅は、16ビットではなく、32ビットです。したがって、個々のセグメント長は、リアルモードでは64キロバイトであるのに対して、386 ネイティブモードでは、最大長4ギガバイトです。
3) リニアアドレス幅（および物理アドレス幅）は、リアルモードでは20ビットであるのに対して、386 ネイティブモードでは32ビットです。したがって、386 ネイティブモードでは最大4ギガバイトのメモリを直接アドレス指定できますが、リアルモードでは1メガバイトまでです。
4) 命令のオペランド幅は32ビットです（このことは、特に算術演算においては重要である）。32ビット幅の汎用レジスタを使用できます（付録F参照）。

80386 を386 ネイティブモードで実行するように書かれた簡単なプログラムを次に示します。この例は、上例からとったHELLO.ASM プログラムですが、1ビットを386 ネイティブモードにしてあります。両者の根本的な違いは、セグメントがUSE32 セグメントとしてセットされている（これは80386 の場合はデフォルト値であり、その意味はこの例の後に説明する）ことと、プログラム中の一対の命令が、32ビットレジスタの使用と、80386 用に命令を加えたことを示していることです。

```
TITLE     hello.asm

;
; The .386c directive is actually unnecessary,
; since it is the default for 386 | ASM.
;
          .386c

          ASSUME    cs:cseg, ds:dseg, ss:sseg

;
; We give the segments a use type of USE32 (the
; default for the 80386).
;
data      SEGMENT PARA PUBLIC USE32 'DATA'

hellomsg  DB      'Hello world.....', 0Dh, 0Ah, '$'

data    ENDS

sseg    SEGMENT PARA STACK USE32 'STACK'

        DB      1000 DUP (?)

sseg    ENDS

cseg    SEGMENT PARA PUBLIC USE32 'CODE'

print   PROC    NEAR
;
; currently, programs linked by 386 | LINK to
; execute in protected mode are set up with a
; single segment, and 386 | DOS-Extender initializes
```

```
; all segment registers to point to it. There is
; therefore no need to initialize DS in a
; protected mode program.
;

;
; Although it is unnecessary, we use the 32-bit
; registers and a protected mode instruction here
; to load the MS-DOS function code into AH.
;
        move   al, 09h, ; load function code into AL
        xor    ebx,ebx ; clear EBX to shift in 0's
        shld   eax,ebx,8 ; shift code intl AH
        mov    dx,OFFSET hellomsg
        int    21h        ; call MS-DOS

        mov    ah,4ch
        int    21h        ; terminate process

print  ENDP

cseg   ENDS

        END    print
```

上記のように、80386 の386 ネイティブモード操作をサポートするMS-DOSのバージョンは現在はありません。Phar Lap Software 社は製品386 | DOS-Extenderをリリースしていますが、この製品により、386 ネイティブモードで実行するように書かれたプログラムは、MS-DOSの制御下で実行可能であり、MS-DOSシステムのサービスを完全に活用することができます。詳細は、386 | DOS-Extenderユーザマニュアルを参照してください。

80286 プロセッサもプロテクトモードのオペレーションがあります。これは80386 ネイティブモードと類似しています。GDT とLDT があり、そのセグメントセレクタは同じです。しかし80286 では、セグメントオフセットの幅はまだ16ビットのままです。セグメントベースアドレスの幅、およびセグメントセレクタとオフセットから形成したリニアアドレスおよび物理アドレスの幅は24ビットであり、80286 プロテクテッドモードの16メガバイトアドレス空間を与えます。

インテルは8086ファミリでは上位互換性を政策としてとっています。その政策の一部として80386 は、80286 プロテクトモード用に書かれたプログラムを修正なしで実行する方

法を持ちます。セグメントには使用型の概念（3.7.3 項参照）が与えられます。セグメントは、USE16 セグメントまたはUSE32 セグメントとして指定され、そのセグメント用のディスクリプタテーブルエントリにその旨が表示されます。

USE32 セグメントはデフォルト値であり、上記のように動作する。USE16 セグメントは、80286 と互換性の維持のために備わっています。USE16 セグメント内のオフセット幅は、16ビットなので最大セグメントサイズは64KBです。命令オペランドの幅は、デフォルト値であるUSE32 セグメントの場合の32ビットではなく、16ビット幅とみなされます。USE16属性を持つスタックセグメントは、PUSHs およびPOPsで、ESP レジスタではなくSPレジスタを使います。それでもUSE16 セグメントから32ビット幅のレジスタセットを使い、幅32ビットのメモリオペランドを処理することはできます。しかし、それを実行する命令はオーバーライドバイトによってプレフィクスされなければなりません。386｜ASM は必要に応じて自動的に、USE16 セグメント中にオペランドサイズとアドレスサイズのオーバーライドバイトを生成します。

通常、USE16 セグメントは80386 ネイティブモード用プログラムと共に使ってはいけません。このセグメントは、もともと80286 プロテクトモード用に書かれ、そのあと80386に移植されたプログラムにだけ、使用されます。残念なことに、USE16 とUSE32 のセグメントが混ざったプログラムを書くこともできます。これは勧められませんが、行う場合には、USE16 とUSE32 のセグメントの間の制御トランスファを行う上で守るルールがあります。詳細は付録Ｊを参照してください。

11.3 80386 をリアルモードで使用する方法

80386 がリアルモードで実行している時でも、80386 用に追加された32ビットレジスタセットと新しい命令を使うことは可能です。32ビットレジスタへのアクセスとアドレスモードは、オーバーライドバイトを使って命令をプレフィックスすることにより行います。新しい命令は、リアルモードの実行時でも、プロセッサによって自動的に認識実行されます。リアルモードにおける最大セグメントサイズ64KBの制限を避ける方法はないことに注意してください。

386｜ASM に80386 命令をアセンブルさせるには、プログラムをターゲットマシン80386 （デフォルト値）でアセンブルします。32ビット命令オペランド用のオーバーライドバイトを生成させるには、USE16 セグメントを指定します。プログラムは、リアルモードの場合に通常行うように、ターゲットマシン8086でリンクします。したがって、リアルモードで実行するようにプログラムのアセンブルとリンクは行いますが、32ビットレジスタセットと新しい命令セットを使うためには、次のコマンドを入力します。

```
386asm hello
386link hello - 8086
```

ここで再び前掲のプログラム例を変更して、32ビットレジスタなどを、リアルモードのプログラムからアクセスする方法を示します。次のプログラムと386 ネイティブモードプログラム例とのただ一つの違いは、USE16 セグメントを使用しているので、リアルモードで実行できるコードが生成されることです。

```
TITLE     hello.asm

.386c     ; assemble for protected mode

ASSUME    cs:cseg, ds:dseg, ss:sseg

;
; We give the segmentes a use type of USE16, so
; that the assembler will generate override bytes
; to allow us to access 32-bit operands.
;
data   SEGMENT PARA PUBLIC USE16 'DATA'

hellomsg DB    'Hello world.....', 0Dh, 0Ah, '$'

data   ENDS

sseg   SEGMENT PARA STACK USE16 'STACK'
       DB      1000 DUP (?)

sseg   ENDS

cseg   SEGMENT PARA PUBLIC USE16 'CODE'

print  PROC    NEAR

       mov     ax, dseg
       mov     ds, ax        ; Init DS
;
; As in the protected mode program, we use 32-bit
; registers and the new instruction SHLD.  However,
; the code generated will be different;          since we
; are in a USE16 segment, we will get operand size
; override bytes for the 32-bit operands.
;
```

```
        mov     al,09h      ; load function code into AL
        xor     ebx,ebx     ; clear EBX to shift in 0's
        shld    eax,ebx,8   ; shift code into AH
        mov     dx,OFFSET hellomsg
        int     21h         ; call MS-DOS

        mov  ah,4ch
        int  21h            ; terminate process

print  ENDP

cseg   ENDS

       END      print
```

付録A 386|ASM コマンドラインスイッチ

386ASM	file1,file2,...,filen	入力ファイル名。
	-LIST file	ファイル名をリストする。
	-NOLIST	リスティングファイルを作成しない。
	-ERRORLIST file	エラーリスティングファイルを作成する。
	-OBJECT file	オブジェクトファイル名。
	-NOOBJECT	オブジェクトファイルを作成しない。
	-8086 -80186 -80286 -80386	ターゲットCPU を選択する。
	-80286P -80386P	保護命令でCPU を選択する。
	-N087 -8087 -80287 -80387	ターゲット演算コプロセッサを選択する。
	-INCLUDE path	INCLUDE ディレクティブパスをセットする。
	-DEFINE name [=string]	テキストシンボルを定義する。
	-NOSYM	リスティング中のシンボルテーブルを作成しない。
	-NODELETE	エラーが発生してもオブジェクトファイルを作成する。
	-ONECASE -TWOCASE	シンボルの大文字／小文字指定をセットする。
	-FULLWARN	特殊エラーチェックを可能にする。

付録B　エラーメッセージ

プログラムのアセンブル時にエラーが発生すると、386｜ASM は画面上にエラーメッセージを表示します。エラーメッセージにはエラーが発生したソースコードファイル、そのファイル内の行番号、エラー識別番号およびエラーメッセージが明示されます。同じメッセージが.LSTファイルにも書き込まれます。

アセンブラエラーは次の3種類に分類されます。(1) 1000未満のエラー番号は、警告を明示するのに使われます。警告とは、アセンブラがそのエラー文に対して無効なオブジェクトコードをかならずしも生成するとはかぎらないエラーです。(2) 1001から2000未満のエラー番号は、重大エラーを明示するのに使われます。重大エラーとは、アセンブラに、そのエラー文に対する正しいオブジェクトコードの生成を不可能にさせるエラーです。(3) 2001以上の番号のエラーは、致命的エラーです。致命的エラーは、アセンブラに、そのソースコードファイルの残り部分を処理することなく直ちに実行を終了させます。アセンブリのパス2で起こった致命的エラーは、ただリスティングファイルに書き込まれるだけです。

以下の3節は、386｜ASM で発生する警告、重大エラー、致命的エラーをリストします。エラーに対する、起こりうる原因と可能な処置も示します。エラーの表示は番号順に行います。

B.1　－　警告

WARNING 1:　Syntax error - name ignored by directive

原因：　名前を使わないディレクティブにユーザ定義名を与えた。

WARNING 2:　Can't close listing file: filename

WARNING 3:　Can't close object file: filename

WARNING 4:　Can't close error list file: filename

WARNING 5:　Can't close cross-reference file: filename

原因：　1．ディスクがいっぱいである。

　　　　2．MS-DOS常駐コードが破壊された。

処置：　1．ディスクに空きスペースを作るためファイルを削除する。

　　　　2．システムを再ブートする。

WARNING 6:　Can't close source file: filename

WARNING 7:　Can't close include file: filename

原因：　MS-DOS常駐コードが破壊された。

処置：　システムを再ブートする。

WARNING 8:　Source line too long - truncated

原因：　ソースラインが132 文字を超えたため、132 文字までに切り捨てられた。

処置：　ソースコードラインを短くする。

WARNING 9;　at character number n: Illegal printing character - ignored

原因：　アセンブラが認識する文字セット以外の文字があったため、スペースとして扱った。

WARNING 10：　at character number n: Illegal non-printing character-ignored

原因：　スペース、復帰、改行、水平タブ、用紙送り以外のコントロールコードがあったため、スペースとして扱った。

WARNING 11:　at character number n: character used in illegal context-ignored

原因：　文字が正しくない文脈で使われているので、スペースとして扱った。

WARNING 13:　String not terminated

原因：　文字列定数上に終止引用符がない。

WARNING 14:　Missing ENDIF for conditional block

原因：　条件文ブロックを終了させるのに必要なENDIF がない。

WARNING 15:　Missing ENDM for macro definition

原因：　マクロ定義またはリピートブロックを終了させるのに必要なENDMがない。これによりソースファイル中の残り行がすべてマクロ定義の一部として扱われた。

WARNING 16:　Integer overflow

原因：　定数または定数式がデータタイプに対して大きすぎる。

処置：　データタイプを変えるか、定数にもっと小さい値を与える。

WARNING 17:　Floating point overflow

原因：　浮動小数点定数がデータタイプに対して大きすぎる。

処置：　データタイプを変えるか、定数にもっと小さい値を与える。

WARNING 18:　Floating point underflow

原因：　浮動小数点定数がデータタイプに対して小さすぎる。

処置：　データタイプを変えるか、定数にもっと大きい値を与える。

WARNING 19:　Decimal point following hexadecimal digits

原因：　小数点は10進数字だけに使用できる。

WARNING 20:　Macro purged or redefined within itself: macroname

原因：　マクロが消去されたか再定義された。これはMicrosoft と互換性維持の点では誤りではないが、幾分曖昧な慣行である。

処置：　警告を無視するかマクロ定義を修正する。

WARNING 21:　Digits too large for radix - base 16 assumed

原因：　定数中の１桁または複数桁が、カレントデフォルト基数、または定数の終りの基数指定子文字で指定された基数に対して大きすぎた。

処置：　数を修正するか定数の終りに基数指定子を加える。

WARNING 22:　Unmatched ENDM

原因：　ENDMが、先行するMACRO 、RP、IRPC、またREPTなしに用いられた。

WARNING 23:　Extra characters on line

原因：　ディレクティブ用または命令用のオペランドを処理したあとで、まだ文字がソースラインに残っていた。

WARNING 24:　Illegal use of symbol in expressions: symbol

原因：　ユーザ定義シンボルかアセンブラ予約語を式中に誤って使用した。

WARNING 25:　Divide by zero

原因：　式中でゼロ除算をしようとした。

WARNING 26:　Reference to multiply defined symbol: symbolname

原因：　文が、ソースコードファイルの別の場所で複数回定義されたシンボルを参照している。

処置：　シンボル定義は一つだけ残してあとはすべて取り除く。

WARNING 27:　No IF to mach ELSE or ENDIF

原因：　有効な条件付きブロックがない時にELSE文またはENDIF 文が用いられた。

WARNING 28:　Two ELSEs in a row

原因：　あいだにENDIF 文またはIF文がなく、二つのELSEディレクティブが連続して用いられた。

WARNING 29:　Floating point denormal value generated

原因：　浮動小数点数が小さすぎて、指定データタイプにとっては異常な値である。

WARNING 30:　Constant definition will not retain its data type

原因：　等号（=）ディレクティブのあとに続く絶対式はデータタイプを持っていたが、無視された。そのディレクティブの前にあるシンボル名が、データタイプを持たない絶対定数としてシンボルテーブルに入力された。

処置：　絶対定数が関連データタイプを持つ必要のない場合、式からデータ指定子を取り除く。データタイプの必要な場合、定数は、代わりにEQU ディレクティブを使うことによって、テキスト置換としてシンボルテーブルに入力できる。

WARNING 31:　Instruction shortened, NOPs inserted

原因：　命令内の前方参照によって、パス２で必要とするよりも多くのスペースがパス１で割当てられた。未使用スペースはNOP 命令で埋められた（このエラーは、-FULLWARN スイッチ使用時にだけ報告される）。

処置：　警告原因となった前方参照を取り除くか、PTR またはセグメントオーバーライド演算子またはその両方を使って、前方参照されたシンボルについてより多くの情報をアセンブラに与える。

WARNING 32:　Default segment register has no assumption

原因：　間接アドレスモードでは、オペランドの実効アドレスの計算時に、セグメントレジスタがベースとして使われる。アドレスモードによって示されたセグメントレジスタが何の仮定も持っていない場合、386 | ASM は警告のフラグをたて、プログラマに仮定を持たないセグメントレジスタが使われたことを知らせる。

処置：　命令の実行時、ASSUMEディレクティブを使ってセグメントレジスタの持つ値をアセンブラに知らせる。

WARNING 33:　Value combined with segment selector ignored

原因：　セグメントセレクタ値（セグメント名またはSEG 式演算子を使って得た）を定数値と組合わせた。実行のためのプログラムロード時、その組合わされた値はセグメントセレクタ値に置換えられ、その定数値は無視される。

B.2 ー 重大エラー

ERROR 1002: Syntax error

原因： 文に文法エラーがあった。

ERROR 1003: Syntax error - instruction expected

原因： 命令ラベルのあとに命令以外のものが続いた。

ERROR 1004: Syntax error - directive expected

原因： ユーザ定義シンボルのあとにディレクティブ以外のものが続いた。

ERROR 1005: Syntax error - missing argument

原因： ディレクティブに必要な引数がない。

ERROR 1006: Can't open include file: filename

原因： １．無効ファイル名

２．ファイルへのアクセスが拒否された。

処置： １．ファイル名の綴りを直す。

２．アクセス問題をクリアする。

ERROR 1007: Can't open include file, path too long: filename

原因： INCLUDE 文中に指定されたファイル名とコマンドライン中に指定されたパス名の合計が、255 文字を超えた。

処置： パス名をもっと短くするようにインクルードファイルを移動させる。

ERROR 1008: Syntax error - missing name for directive

原因： ユーザ定義名を必要とするディレクティブに名前がない。

ERROR 1009: Illegal statement - no current segment

原因： セグメント内だけで有効とされる文が、セグメントがオープンしていない時に用いられた。すべての命令とディレクティブの多くはセグメント内になければならない。

ERROR 1010: Symbol already a reserved word: symbolname

原因： アセンブラ予約語の一つと同じ名前でユーザ定義シンボルを作成しようとした。

ERROR 1011: Symbol already different kind: symbolname

原因： 前に定義したシンボルを再定義しようとした。

ERROR 1012: Segment parameters are changed

原因： 定義のために使ったセグメント属性とは別のセグメント属性で、そのセグメントを再オープンしようとした。

ERROR 1013: Count for DUP operator cannot be relocatable

原因： DUP 演算子におけるカウントパラメータには定数値だけが使用できる。

ERROR 1014: Improper segment attribute specifier.

原因： セグメントディレクティブのあとに続く一つまたは複数のキーワードが有効なセグメント属性指定子ではない。これはアライン型キーワード、組合せ型キーワード、アクセス型キーワード、または使用属性キーワードでなければならない。（例：BYTE, PUBLIC, ER, USE16）

ERROR 1015: Block nesting error.

原因： 既にオープンしているセグメントを再オープンしようとしたか、現在オープンしていないセグメントまたは構造体に対してENDSディレクティブが用いられた。

ERROR 1016: Open procedure: procname

原因： procnameという名のプロシジャをPROCディレクティブでオープンしたが、対応するENDPディレクティブでクローズしなかった。

ERROR 1017: Undefined symbol: symbolname

原因： 未定義のシンボルを参照した。

ERROR 1018:　Open segment: segname

原因：　segname という名のセグメントをSEGMENT ディレクティブでオープンしたが、対応するENDSディレクティブでクローズしなかった。

ERROR 1019:　Symbol multiply defined: symbolname

原因：　既に定義されているシンボルを再定義しようとした。

ERROR 1020:　Segment already in a different group: segname

原因：　セグメントsegname を複数のグループに入れようとした。

処置：　segname を一つだけ残して残りのすべてのグループ定義から取り除く。

ERROR 1021:　Syntax error - invalid expression

原因：　式が、正しい式作成ルールに従っていない。

処置：　演算子優先順位テーブルをチェックし、入力ミスを探す。

ERROR 1022:　DUP operator requires count on left

原因：　データ宣言文ではDUP 演算子の左に絶対カウントが必要である。

ERROR 1023:　DUP operator nested deeply

原因：　データ宣言ディレクティブにおけるDUP 演算子のネストレベルの深さが17より大であった。

処置：　もっと簡単なデータ宣言方法を探す。

ERROR 1024:　Unknown symbol type specifier

原因：　シンボル型キーワード（例：BYTE、NEAR）の必要なディレクティブがそのキーワードを見つけなかった。

ERROR 1025:　Type illegal in context

原因：　ディレクティブと一緒に使用したシンボル型キーワードが、この文脈では有効ではない。

ERROR 1026:　No or unreachable CS

原因：　CSレジスタが異なるセグメントを指定することを仮定している時に、NEARプロシジャを定義した。

処置：　ASSUMEディレクティブを使ってNEARプロシジャの定義前にCSがカレントセグメントを指定することをアセンブラに伝える。

ERROR 1027:　Procedure not open: procname

原因：　現在オープンされていないプロシジャに対するENDPディレクティブがあった。

処置：　不用なENDPディレクティブを取り除く。

ERROR 1028:　Procedure nesting error, closing: procname

原因：　最後にオープンしたものではないプロシジャに対するENDPディレクティブがあった。そのため最後にオープンしたプロシジャがクローズされた。

処置：　各プロシジャに対するPROC/ENDP ディレクティブを正しい順序にする。

ERROR 1029:　Procedure closed in different segment: procname

原因：　名前つきプロシジャのオープンとクローズが別々のセグメントで行われた。

ERROR 1030:　Constant was expected

原因：　定数値だけが許されている時に、リロケータブルな値を使った。

ERROR 1031:　Module already has title

原因：　TITLE ディレクティブが2つあった。TITLE ディレクティブは1モジュールにつき1つだけ許される。

ERROR 1032:　Can't use EVEN on byte segment

原因：　バイトアライン型を持つセグメントでEVENディレクティブが使われた。アセンブラは、セグメントが偶数バイト境界にあるか、奇数バイト境界にあるかが分からないので、埋め込みバイトを生成すべきかどうかを決定できない。

処置：　セグメントのアライン型を変えるか、EVENディレクティブを取り除く。

ERROR 1033:　Value is out of range

原因：　文中に明示的または暗黙的に指定されたデータタイプに対して、値が大きすぎるか、または小さすぎる。

ERROR 1034:　Illegal forward reference to symbol: symbolname

原因：　指定されたシンボルへの前方参照が違法である。その理由はパス1で、アセンブラがそのシンボルについて行った仮定が原因で、パス2で必要とする長さより少ないオブジェクトコードが生成された。

処置：　セグメントオーバーライドかPTR などの演算子を使って、前方参照されるシンボルについてアセンブラに、より多くの情報を与えるか、シンボル宣言を参照文の前に移動させる。

ERROR 1035:　Syntax error - bad data type in expression

原因：　整数以外の定数データタイプの一つ（例：パック10進数、実数、クワッドワード整数）を、間違ったデータ宣言ディレクティブ（DB、DW、DQ、DP、DT）と共に使用した。

ERROR 1036:　Symbol already defined external: symbolname

原因：　モジュール中で既に参照した外部シンボルをローカルに定義しようとした。外部シンボルのローカル再定義は、そのシンボルが未使用の場合にだけ行うことができる。

処置：　ローカル定義を取り除くか、その上のすべての参照を取り除く。

ERROR 1037:　Symbol not defined external: symbolname

原因：　シンボルを一つのモジュール中でパブリックシンボルと外部シンボルの両方に宣言したので、パブリック定義が優先された。

処置：　外部宣言を取り除く。

ERROR 1038:　Symbol already defined locally: symbolname

原因：　すでにローカルに定義したシンボルを外部シンボルとして宣言しようとした。

処置：　そのシンボルのEXTRN 定義を取り除く。

ERROR 1039: Local and external definitions differ for: symbolname

原因： シンボルの外部宣言で与えられたシンボルデータタイプがそのローカル定義に合致しない。

ERROR 1040: Symbol type illegal for PUBLIC: symbolname

原因： 変数、ラベル、または定数以外のシンボルを、PUBLIC定義で使った。

ERROR 1041: Segment override is illegal

原因： 文脈においてセグメントオーバーライドを使うことは誤りである。

ERROR 1042: Improper operand type

原因： 命令またはディレクティブに対してオペランドのデータタイプが誤りである。

ERROR 1043: Operand types don't match

原因： 命令またはディレクティブの複数のオペランドが持つデータタイプに互換性がない。

ERROR 1044: Circular text substitution: symbolname

原因： 指定されたシンボル名が、テキスト置換シンボルであって、このシンボルを展開すると循環的なテキスト置換の原因となる。

ERROR 1045: Multiply defined symbol: symbolname

原因： シンボルをソースプログラム中の複数の場所で定義した。

ERROR 1046: Missing)

原因： 式中に“（”と対になる“）”がない。

ERROR 1047: Missing]

原因： 式中に“［”と対になる“］”がない。

ERROR 1048:　Invalid use of relocatable operand

原因：　リロケータブルな値と式演算子との組み合わせ方法が誤りであった。

ERROR 1049:　Invalid use of segnent override

原因：　式が、別々のセグメントへオーバーライドする二つの値を組み合わせようとした。

ERROR 1050:　Incompatible data types in expression

原因：　式が、異なるデータタイプを持つ二つの値を組合せようとした。

ERROR 1051:　Illegal use of register

原因：　式中でレジスタ名を誤って使った。

ERROR 1052:　Illegal constant value for expression

原因：　整数以外の定数データタイプ（例：パック10進数、実数、クワットワード整数）の一つが、単項演算子＋または－以外の演算子と一緒に使われた。

ERROR 1053:　Operand must have size

原因：　アセンブラは、正しいオブジェクトコードを生成するために、命令に対するオペランドのサイズを知らなければならない。

処置：　PTR 演算子でオペランドのサイズを指定する。

ERROR 1054:　Illegal addressing mode

原因：　命令に対して不正なアドレスモードを使った。

ERROR 1056:　Can't reach with segment register

原因：　現在仮定されているセグメントレジスタ値のいずれを使ってもメモリ参照オペランドを参照できない。

処置：　実行時ASSUMEディレクティブを使って、どのセグメントレジスタがオペランドのセグメントを指定するかをアセンブラに伝える。

ERROR 1057: Illegal scale factor index register

原因： 1、2、4、8以外のスケールファクタをインデクスレジスタで使った。

ERROR 1059: ST(n) 8087 register out of range

原因： 8087浮動小数点スタックレジスタの一つを識別するのに、0-7 の範囲外の数値を使った。

ERROR 1060: Text substitution (from EQU) caused buffer overflow - line truncated

原因： ソース文が、一つまたは複数のテキスト置換シンボルへの参照を持ち、このシンボルを展開すると、アセンブラの内部バッファをオーバーフローさせる。文の内部表示を強制的に切り捨てる。

処置： その文を複数の文に分解する。

ERROR 1061: Near JMP/CALL to differeat CS

原因： near JMPまたはnear CALL を別セグメントのラベルに試みた。

ERROR 1062: AT combine balue must be last on line

原因： SEGNENT ディレクティブで、AT組合せ型指定子のあとには、セグメント属性を与えることはできない。

ERROR 1063: Relative value out of range

原因： PC相対値が127 より大きいか、-128より小さい。

ERROR 1064: Wrong number of digits in hexadecimal real

原因： 16進数表示の実数が、8、16、20 桁のいずれでもない。

ERROR 1065: Illegal packed decimal number

原因： パック10進数が19桁以上であるか、基数10に対して大きすぎるディジット（たとえばA-F）を持っている。

ERROR 1066:　Syntax error- no expression following % operator

原因：　マクロ呼出し文中の% 演算子のあとに有効な式が続いていなかった。

ERROR 1067:　Missing END directive added by assembler

原因：　ソースファイルにEND 文がなかった。アセンブラが、END 文をアセンブリの最終ソースファイルの最終行のあとに挿入した。

ERROR 1068:　Statement only valid inside macro

原因：　マクロ定義の外側にLOCAL ディレクティブまたはEXITM ディレクティブがあった。

ERROR 1069:　SHORT operator only valid for label

原因：　SHORT 式演算子を、命令ラベル以外の値と一緒に使った。

ERROR 1070:　Missing >

原因：　“＜”と対になる“＞”がない。

ERROR 1071:　Cannot override ES register

原因：　文字列の宛先オペランドはESレジスタと互換性がなければならない。

処置：　ASSUMEディレクティブまたはセグメントオーバーライド演算子を使って、確実にESレジスタでそのオペランドに到着できるようにする。

ERROR 1072:　Long parameter caused line truncation in macro expansion

原因：　マクロ展開でパラメータ置換を行ったため、133 文字以上のソースコードラインを生成した。ソースラインは、132 文字までに切り捨てられた。

ERROR 1073:　IF directives nested too deeply

原因：　IFディレクティブのネストの深さが255 レベルを超えた。

ERROR 1074:　Macros nested too deeply

原因：　マクロ展開のネストの深さが65,535レベルを超えた。

ERROR 1076:　Missing ENDS for structure definition

原因：　構造体定義ブロックの終了に必要なENDSディレクティブがない。

ERROR 1077:　Forced error

原因：　.ERRディレクティブによってエラーが強制的に発生させられた。

ERROR 1078:　Forced error - expression equals 0

原因：　.ERREディレクティブによってエラーが強制的に発生させられた。

ERROR 1079:　Forced error - expression not equals to 0

原因：　.ERRNEディレクティブによってエラーが強制的に発生させられた。

ERROR 1080:　Forced error - symbol defined

原因：　.ERRDEFディレクティブによってエラーが強制的に発生させられた。

ERROR 1081:　Forced error - symbol not defined

原因：　.ERRNDEFディレクティブによってエラーが強制的に発生させられた。

ERROR 1082:　Forced error - string blank　:

原因：　.ERRBディレクティブによってエラーが強制的に発生させられた。

ERROR 1083:　Forced error - string not blank

原因：　.ERRNBディレクティブによってエラーが強制的に発生させられた。

ERROR 1084:　Forced error - string identical

原因：　.ERRIDNディレクティブによってエラーが強制的に発生させられた。

ERROR 1085:　Forced error - string different

原因：　.ERRDIF ディレクティブによってエラーが強制的に発生させられた。

ERROR 1086:　Cannot change target CPU after a segment definition

原因：　セグメントのオープン後に、.8086、.186、.286、.286c、.286p、.386、.386c、.386p、または.PROT を使用しようとした。

ERROR 1087:　Segment access types must form a compatible set

原因：　組合せできないアクセス型を持つセグメントを定義しようとした。

　例：　ERとRW

ERROR 1088:　Module already has a name

原因：　同一ソースプログラム中にNAMEデレクティブが二つあった。

ERROR 1089:　Statement illegal inside structure definition

原因：　構造体定義ブロックは現在オープンしているが、構造体定義内の文が正しくない。構造体定義内に入れることができるのは、データ宣言文とコメント文だけである。

ERROR 1090:　Field cannot be overriden

原因：　構造体の作成時、初期化できない構造体フィールドを初期化しようとした。多重値を持つフィールドはすべて初期化できない。

ERROR 1091:　Override value is too long

原因：　構造体フィールドの初期化に使用された定数文字列が長すぎて、フィールドにあわない。

ERROR 1092:　Override with DUP is illegal

原因：　構造体フィールドまたはレコードフィールド用のイニシャライザが、式中にDUP 演算子を持っていた。DUP 演算子は、イニシャライザの式中で使用することは違法である。

ERROR 1093:　More values than defined fields

原因：　構造体中またはレコード中のフィールド数よりも、構造体宣言またはレコード宣言用のカギ括弧内の値のほうが多い。

ERROR 1094:　Override with string is legal on byte fields only

原因：　定数文字列をBYTEのデータタイプを持たないフィールドのイニシャライザとして使用しようとした。

ERROR 1095:　Use of structure requires initializer list

原因：　構造体をイニシャライザリストを指定しないで作成しようとした。リストが空でもカギ括弧は必要である。

ERROR 1096:　Field width specifier must be adsolute constant　:

原因：　定義中のレコードフィールドの幅指定子は、絶対定数であることが必要である。

ERROR 1097:　Field value must be adsolute constant

原因：　レコードフィールド値は、絶対定数でなければならない。

ERROR 1098:　Default field value is too large for field width :

原因：　作成中のレコード内フィールドのデフォルト値が大きすぎて、そのフィールドに指定された幅に合わない。

ERROR 1099:　Initial field value is too large for field width

原因：　レコードフィールドのイニシャライザが大きすぎて、初期化中のフィールドに合わない。

ERROR 1100:　Total size of record fields is too large

原因：　作成中のレコード内フィールドの合計サイズが、32ビットより大きい（または16ビットより大きくターゲットCPU が80386 ではない）。

ERROR 1101:　Use of record requires initializer list

原因：　イニシャライザリストを指定しないでレコードを作成しようとした。リストが空でもカギ括弧はつけなければならない。

ERROR 1102:　String illegal for record field override

原因：　作成中のレコード内のフィールドのイニシャライザとして定数文字列を使用しようとした。

ERROR 1103:　Local symbol not allowed with directive

原因：　プロシジャブロックのローカルシンボル（"#" で始まるシンボル）は、このディレクティブでは定義できない。ローカルシンボルの作成にはデータ宣言ディレクティブEQU 、LABEL 、および＝だけが、使用できる。

ERROR 1104:　Local symbol must be defined inside a procedure

原因：　ローカルシンボル（"#" で始まるシンボル）の定義は、プロシジャブロック内だけで行うことができる。

ERROR 1105:　Align directive requires a power of two

原因：　アラインディレクティブに対するオペランドは、２の累乗でなければならない。そうしないとロケーションカウンタは変更されない。

ERROR 1106:　Descriptor table register requires PWORD operand

原因：　ディスクリプタテーブルレジスタ（IDTRとGDTR）上で動作する命令には、PWORD のデータタイプを持つメモリ参照オペランドが必要である。

処置：　PTR 演算子は、必要ならばメモリ参照オペランドがデータタイプPWORD を持つよう強制するのに使われる。

ERROR 1107:　Location counter has wrapped

原因：　現在オープン中のセグメントのロケーションカウンタがオーバーフローして回り込んだ。これにより、セグメントの頭の初期化されたデータは破壊され、セグメントのサイズは、リンク時に不正確になる。

ERROR 1108:　Count for DUP operator cannot be a forward reference

原因：　DUP演算子と一緒に使用するパラメタは、定数値でなければならず、またソースファイル中にあとで定義されるシンボルへの参照を持つことはできない。

ERROR 1109:　Relocatable ORG value must lie in current segment　:

原因：　ORGディレクティブ値が、現在アセンブル中のセグメントとは異なるセグメントに関連してリロケートされた。386 | ASM では、ORG に対するオペランドを、絶対値にするか現在オープン中のセグメントとの関連でリロケートすることが必要である。

B.3　一　致命的エラー

FATAL ERROR 2001:　System error

原因：　386 | ASM にバグがある。エラー原因のソースコードファイルのコピーを保存して、当社に連絡すること。

FATAL ERROR 2002:　Can't create listing file: filename
FATAL ERROR 2003:　Can't create object file: filename
FATAL ERROR 2004:　Can't create error list file: filename
FATAL ERROR 2005:　Can't create cross-reference file: filename

原因：
1. ファイル名かそのディレクトリ名のスペルを間違って入力した。
2. ファイルを置くディレクトリが存在しない、またはそのディレクトリに書込みアクセスをしていない。
3. ディスクがいっぱいである。

処置：
1. ファイル名またはディレクトリ名を正しく入力する。
2. アクセス問題を解決する。
3. ディスクの領域をあけるために、使わないファイルを処分する。

FATAL ERROR 2006:　Error writing to listing file: filename
FATAL ERROR 2007:　Error writing to object file: filename
FATAL ERROR 2008:　Error writing to error list file: filename
FATAL ERROR 2009:　Error writing to cross-reference file: filename

原因：　ディスクがいっぱいである。

処置：　ディスクの領域をあけるために、使わないファイルを処分する。

FATAL ERROR 2010:　Can't open source code file: filename

原因：　1．ファイル名かそのディレクトリ名のスペルを間違って入力した。

　　　　2．そのファイルまたはそのディレクトリに読取りアクセスしていない。

処置：　1．ファイル名またはディレクトリ名を正しく入力する。

　　　　2．アクセス問題を解決する。

FATAL ERROR 2011:　Error reading source code file: filename

原因：　ファイル損傷の可能性がある。

処置：　ファイルを作成し直すか、バックアップコピーからファイルを復元する。

FATAL ERROR 2012:　Out of memory

原因：　386|ASM はシンボルテーブルをメモリに格納する領域を使い尽くした。

処置：　ソースコードファイルを複数ファイルに分割し、それを別々にアセンブルして、386|LINKで組み合わせる。

FATAL ERROR 2013:　Symbol changed offset on pass 2

FATAL ERROR 2014:　Procedure changed size on pass 2

FATAL ERROR 2015:　Segment changing size on pass 2

原因：　386|ASM にバグがある。エラー原因のソースコードファイルのコピーを保存して、当社に連絡すること。

FATAL ERROR 2016:　Illegal symbol name difined on command line: symbolname

原因：　コマンドラインに-DEFINE スイッチで定義したシンボル名が、正しいシンボル名構成のルールに従っていない。

付録C 形式エレメント

C.1 ー 文字セット

386｜ASM は次の文字セットを認識します。

A-Z
a-z
0-9
+ - * /
() [] <>
' "
@ ? $ #
, ; . : =
% ! &
スペース　　復帰　改行　水平タブ

386｜ASM はこの文字セット以外の文字をスペースとして扱います。

C.2 ー 文

386｜ASM はソースコードファイルを1行ずつ処理します。1行は、改行文字、または1対の復帰／改行文字によって終了します。ソースコードラインの長さは132 文字以下です。

386｜ASM は各ソースコードラインを解析して、トークンに分解します。トークンには、次の種類があります。区切り文字、アセンブラ予約名、識別子（ユーザ定義名）、定数値、および文字列定数です。区切り文字は常に単一のソース文字です。他のトークンの長さはすべて単一または複数のソース文字であり、トークンタイプ用の形式ルールに従わない文字によって終了します。

文はソースファイルの単一行に書きます。文の一般的形式を次に示します。

```
{identifer } mnemonic {operand,... }  {;comment}
```

括弧内の項目はオプションであり、mnemonicは命令ニモニックまたはディレクティブ名です。

C.3 ― アセンブラ予約名

予約名は、386|ASM に対して特別の意味をもちます。予約名は再定義できません。予約名には、命令、ディレクティブ、レジスタ名、式演算子などがあり、これらについてはすべて、本マニュアルの個別の付録に表示します。さらに、表C－1に表示した名前は予約名です。大小文字を区別したアセンブルが有効無効にかかわらず、予約名の大文字、小文字はすべて同じに取り扱われます。

表C－1　その他の予約名

名　前	80386 のみ	説　明
ABS	以外も有	EXTRN ディレクティブと一緒に使用するデータタイプ
BYTE	以外も有	変数データタイプ
DWORD	以外も有	変数データタイプ
DUP	以外も有	値の複写を指定するためデータ宣言と一緒に使用する演算子
EO	有	セグメントアクセス型属性
ER	有	セグメントアクセス型属性
FAR	以外も有	ラベルデータタイプ
FWORD	有	変数データタイプ
NEAR	以外も有	ラベルデータタイプ
NOTHING	以外も有	ASSUMEディレクティブ用キーワード
PARA	以外も有	セグメントアライン型
PWORD	有	変数データタイプ
QWORD	以外も有	変数データタイプ
RO	有	セグメントアクセス型属性
RW	有	セグメントアクセス型属性
TBYTE	以外も有	変数データタイプ
USE16	有	セグメント使用属性
USE32	有	セグメント使用属性
WORD	以外も有	変数データタイプ

C.4 － 識別子

識別子はユーザ定義名です。識別子は次の文字の任意の組合せです。

A-Z

a-z

0-9

_ ? $ @

識別子の先頭は数字にすることはできません。識別子は最初の31文字だけが有効であり、残りはすべて無視されます。コマンドラインスイッチで大小文字区別アセンブルを有効にしない限り、同一識別子の大文字小文字は、386 | ASM によって同じに扱われます。

また、ローカル識別子は、記号"#"から始めることによって、プロシジャブロック内で定義できます。ローカル識別子は、定義されているプロシジャブロック内でのみ認識されます。

C.5 － 区切り文字

区切り文字は、時には文脈を区別したりして、386 | ASM に対して特別な意味を持つ単一文字で構成されるトークンです。386 | ASM によって認識される区切り文字を、表C－2に示します。

表C－2　区切り文字

区切り文字	説　　明
+ － * ／ . : () []	式演算子
< > & ! %	マクロ演算子
;	文のコメント部分の始まり
,	命令またはディレクティブのオペランドを区切る。
$	カレントセグメント内のロケーションカウンタのカレント数値を示す。
' "	文字列定数を囲むのに使う。

C.6 － 定数

整数　digits {radix_specifier}

整数は、デフォルト基数（.RADIXディレクティブを使ってセットできる）における一つまたは複数のディジット文字列です。16進数の入力に使うA-F ディジットは、大文字または小文字で示すことができます。基数指定子（表Ｃ－３）は、デフォルト基数をオーバーライドするために、ディジット文字列に加えることもできます。16進数は、常に10進数（０－９）で始まることに注意してください。英字で始まるトークンは、識別子として扱われます。それを数字として識別することが必要な場合には、16進数の先頭に０をつけます。

表Ｃ－３　基数指定子

基数指定子	基　　　数
B	２進（基数２）
Q	８進（基数８）
O	８進（基数８）
D	10進（基数10）
H	16進（基数16）

実数　digits. {digits} {E {+1-} digits}

実数は、整数部（必須）、小数点（必須）、小数部（オプション）および指数（オプション）から構成されます。すべてのディジットは10進数（０－９）です。実数は、DD、Q 、のディレクティブと一緒にのみ使われます。各ディレクティブに許される実数の範囲については、データタイプについて説明した付録を参照してください。

コード化された実数　digitsR

コード化された実数は８桁（CD）、16桁（DQ）、または20桁（DT）の16進数で入力され、そのあとに実数指定子Ｒが続きます。数字の先頭が０である場合、桁数は一つだけ増やすこともできます。

パック10進数　digits

パック10進数は、8087パック10進形式で記憶された10進数字です。パック10進数は、デフォルト基数が10進数であり、その数字の終りに基数指定子が使われていない場合に、DTディレクティブによって作成されます。パック10進数の最大桁数は18です。

C.7 ー 文字列定数

文字列定数は、一つまたは複数の印刷可能なASCII 文字、NUL(0)を除く印刷不能ASCII 文字、またはダブル引用符(")またはシングル引用符(')で囲まれた非ASCII 文字（ビット７を持つ文字セット）の文字列です。文字列を囲む引用符を、その文字列の一部としてコード化するには、引用符文字を２回タイプします。たとえば、"a""b""c" とタイプすると文字列a"b"c がコード化されます。

８文字までの文字列なら、指定のデータタイプからオーバーフローしないかぎり、整数値として使用できます。文字列中の各文字のASCII 値は、１バイトの整数として、文字の右側が下位のバイトになるように格納されます。

付録D　命令

表D－1　80386 命令セット

命　令	必要なプロセッサ	形　　式	説　　明
AAA	8086	AAA	加算に対してALをASCII 調整する。
AAD	8086	AAD	除算に対してALをASCII 調整する。
AAM	8086	AAM	乗算に対してALをASCII 調整する。
AAS	8086	AAS	減算に対してALをASCII 調整する。
ADC	8086	ADCaccum, immed	キャリーとイミディエイト値をアキュムレータに加える。
	8086	ADCr/m, immed	キャリーとイミディエイト値をオペランドに加える。
	8086	ADCr/m, reg	キャリーとレジスタをオペランドに加える。
	8086	ADCreg, r/m	キャリーとオペランドをレジスタに加える。
ADD	8086	ADDaccum, immed	イミディエイト値をアキュムレータに加える。
	8086	ADDr/m, immed	イミディエイト値をオペランドに加える。
	8086	ADDr/m, reg	レジスタをオペランドに加える。
	8086	ADDreg, r/m	オペランドをレジスタに加える。
AND	8086	ANDaccum, immed	イミディエイト値とアキュムレータとをAND（論理積）する。
	8086	ANDr/m, immed	イミディエイト値とオペランドとをAND（論理積）する。
	8086	ANDr/m, reg	オペランドとレジスタとをAND（論理積）する。
	8086	ANDreg, r/m	レジスタとオペランドとをAND（論理積）する。
ARPL	80286P	ARPLmem, reg	要求された特権レベルを調整。

命　令	必要なプロセッサ	形　　式	説　　明
BOUND	80186	BOUNDreg, mem	範囲外の値を検出する。
BSF	80386	BSFreg, r/m	最下位の１ビットをみつける。
BSR	80386	BSRreg, r/m	最上位の１ビットをみつける。
BT	80386	BT　r/m, reg	レジスタによってインデクスされたビットをテストする。
	80386	BT　r/m, immed	イミディエイト値によってインデクスされたビットをテストする。
BTC	80386	BTCr/m, reg	レジスタによってインデクスされたビットをテストし補数をとる。
	80386	BTCr/m, immed	イミディエイト値によってインデクスされたビットをテストし補数をとる。
BTR	80386	BTRr/m, reg	レジスタによってインデクスされたビットのテストとクリヤを行う。
	80386	BTRr/m, immed	イミディエイト値によってインデクスされたビットのテストとクリヤを行う。
BTS	80386	BTSr/m, reg	レジスタによってインデクスされたビットのテストとセットを行う。
	80386	BTSr/m, immed	イミディエイト値によってインデクスされたビットのテストとセットを行う。
CALL	8086	CALL label	命令をラベルで呼び出す。
	8086	CALL r/m	間接で命令を呼び出す。
CBW	8086	CBW	ALからAXに符号拡張する。
CDQ	80386	CDQ	EAX からEDX:EAX に符号拡張する。
CLC	8086	CLC	キャリーフラグをクリヤする。
CLD	8086	CLD	ディレクションフラグをクリヤする。
CLI	8086	CLI	割込みフラグをクリヤする。
CLTS	80286P	CLTS	タスク切換えフラグをクリヤする。
CMC	8086	CMC	キャリーフラグを反転する。
CMP	8086	CMPaccum, immed	イミディエイト値をアキュムレータと比較する。
	8086	CMPr/m, immed	イミディエイト値をオペランドと比較する。
	8086	CMPr/m, reg	レジスタをオペランドと比較する。
	8086	CMPreg, r/m	オペランドをレジスタと比較する。
CMPS	8086	CMPSstc, dest	文字列を比較する。

命　令	必要なプロセッサ	形　　式	説　　明
CMPSB	8086	CMPSB	BYTE文字列を比較する。
CMPSD	8086	CMPSD	DWORD 文字列を比較する。
CMPSW	8086	CMPSW	WORD文字列を比較する。
CWD	8086	CWD	AXからDX:AX に符号拡張する。
CWDE	80386	CWDE	EAX からEDX:EAX に符号拡張する。
DAA	8086	DAA	加算に対してALを10進調整する。
DAS	8086	DAS	減算に対してALを10進調整する。
DEC	8086 8086	DECr/m DECreg	オペランドから１を引く。 フルサイズレジスタの内容から１を引く。
DIV	8086	DIVr/m	アキュムレータをオペランドで除算する。
ENTER	80186	ENTERimm16, imm8	プロシジャに入る。
ESC	8086	ESC immed, r/m	６ビットイミディエイト値およびオペランドを拡張命令として発行。
HLT	8086	HLT	停止する。
IDIV	8086	IDIVr/m	アキュムレータをオペランドで整数除算する。
IMUL	8086 80186 80286 80386	IMULr/m IMULreg, immed IMULreg, r/m, immed IMULreg, r/m	アキュムレータをオペランドで整数乗算する。 レジスタをイミディエイト値で整数乗算する。 イミディエイト値でオペランドを整数乗算。結果をレジスタに。 オペランドでレジスタを乗算する。
IN	8086 8086	IN accum, immed IN accum, DX	ポートから入力（８ビットイミディエイト値）する。 DXによって与えられたポートから入力する。
INC	8086 8086	INCr/m INCreg	オペランドをインクリメントする。 16ビットレジスタをインクリメントする。
INS	80186	INS mem, DX	ポートDXから文字列を入力する。
INSB	80186	INSB	ポートDXからBYTE文字列を入力する
INSD	80386	INSD	ポートDXからデスティネーション文字列にDWORD を入力する。
INSW	80186	INSW	ポートDXからWORD文字列を入力する

命　令	必要なプロセッサ	形　　式	説　　明
INT 3	8086	INT 3	ソフトウェア割込み３ （ブレークポイント割込み）
INT	8086	INT immed	ソフトウェア割込み0-255
INTO	8086	INTO	オーバーフロー時の割込み
IRET	8086	IRET	割込みからの復帰
IRETD	80386	IRETD	割込みから386 ネイティブドモードに復帰。
JA	8086	JA label	より上(>)であればジャンプする。
JAE	8086	JAE label	以上（≧）であればジャンプする。
JB	8086	JB label	より下(<)であればジャンプする。
JBE	8086	JBE label	以下（≦）であればジャンプする。
JC	8086	JC label	キャリーフラグがセットされていればジャンプする。
JCXZ	8086	JCXZ label	CXレジスタがゼロであればジャンプする。
JE	8086	JE label	等しければジャンプする。
JECXZ	80386	JECXZ label	ECX レジスタがゼロであればジャンプする。
JG	8086	JG label	より大き(>)ければジャンプする。
JGE	8086	JGE label	以上（≧）であればジャンプする。
JL	8086	JL label	未満(<)であればジャンプする。
JLE	8086	JLE label	以下（≦）であればジャンプする。
JMP	8086 8086	JMP label JMP r/m	ラベルで命令にジャンプする。 間接で命令にジャンプする。
JNA	8086	JNA label	より大きく(>)なければジャンプする。
JNAE	8086	JNAE label	以上（≧）でなければジャンプする。
JNB	8086	JNAB label	より下(<)でなければジャンプする。
JNBE	8086	JNBE label	以下（≦）でなければジャンプする。
JNC	8086	JNC label	キャリーフラグがセットされていなければジャンプする。
JNE	8086	JNE label	等しくなければジャンプする。
JNG	8086	JNG label	より大きく(>)なければジャンプする。
JNGE	8086	JNGE label	以上（≧）でなければジャンプする。
JNL	8086	JNL label	未満(<)でなければジャンプする。
JNLE	8086	JNLE label	以下（≦）でなければジャンプする。
JNO	8086	JNO label	ｵｰﾊﾞｰﾌﾛｰﾌﾗｸﾞがｾﾞﾛならジャンプする。
JNP	8086	JNP label	ﾊﾟﾘﾃｨﾌﾗｸﾞ がｾﾞﾛならジャンプする。

命　令	必要なプロセッサ	形　　式	説　　　明
JNS	8086	JNS label	ｻｲﾝﾌﾗｸﾞが ｾｯﾄされていればジャンプする。
JNZ	8086	JNZ label	ｾﾞﾛでなければジャンプする。
JO	8086	JO label	ｵｰﾊﾞｰﾌﾛｰﾌﾗｸﾞが ｾｯﾄされていればジャンプする。
JP	8086	JP label	ﾊﾟﾘﾃｨﾌﾗｸﾞ が ｾｯﾄされていればジャンプする。
JPE	8086	JPE r/m	偶数ﾊﾟﾘﾃｨであればジャンプする。
JPO	8086	JPO label	奇数ﾊﾟﾘﾃｨであればジャンプする。
JS	8086	JS label	ｻｲﾝﾌﾗｸﾞが ｾｯﾄされていればジャンプする。
JZ	8086	JZ label	ｾﾞﾛﾌﾗｸﾞ が ｾｯﾄされていればジャンプする。
LAHF	8086	LAHF	フラグの内容をAHにロードする。
LAR	80286P	LAR reg,mem	アクセス権をロードする。
LDS	8086	LDS reg,r/m	FAR ﾎﾟｲﾝﾀをDSとﾚｼﾞｽﾀにロードする。
LEA	8086	LEA r/m	ｵﾍﾟﾗﾝﾄﾞ の実効ｱﾄﾞﾚｽをロードする。
LEAVE	80186	LEAVE	プロシジャを解放する。
LES	8086	LES reg,r/m	FAR ﾎﾟｲﾝﾀをESとﾚｼﾞｽﾀにロードする。
LFS	80386	LFS reg,r/m	FAR ﾎﾟｲﾝﾀをFSとﾚｼﾞｽﾀにロードする。
LGDT	80286P	LGDT mem	GDT ﾚｼﾞｽﾀにメモリからロードする。(24ビットベース)
LGDTE	80386P	LGDTE mem	GDT ﾚｼﾞｽﾀにメモリからロードする。(32ビットベース)
LGS	80386	LGS reg,r/m	FAR ﾎﾟｲﾝﾀをGSとﾚｼﾞｽﾀにロードする。
LIDT	80286P	LIDT mem	IDT ﾚｼﾞｽﾀにメモリからロードする。(24ビットベース)
LIDTE	80386P	LIDTE mem	IDT ﾚｼﾞｽﾀにメモリからロードする。(32ビットベース)
LLDT	80286P	LLDT mem	ﾛｰｶﾙﾃﾞｨｽｸﾘﾌﾟﾀﾃｰﾌﾞﾙﾚｼﾞｽﾀにロードする
LMSW	80286P	LMSW mem	ﾏｼﾝｽﾃｰﾀｽﾜｰﾄﾞ にロードする
LOCK	8086	LOCK	バスをロックする。
LODS	8086	LODS src	文字列エレメントをロードする。
LODSB	8086	LODSB	BYTEを文字列からALにロードする。
LODSD	80386	LODSD	DWORD をソース文字列から EAXにロードする。
LODSW	8086	LODSW	WORDを文字列からAXにロードする。
LOOP	8086	LOOP label	ラベルで命令にループする。

命　令	必要なプロセッサ	形　　式	説　　　　明
LOOPE	8086	LOOPE label	等しいならラベルで命令に ﾙｰﾌﾟする。
LOOPNE	8086	LOOPNE label	等しくないならラベルで命令にループする。
LOOPNZ	8086	LOOPNZ label	ｾﾞﾛでないならﾗﾍﾞﾙで命令にﾙｰﾌﾟする。
LOOPZ	8086	LOOPZ label	ｾﾞﾛであるならﾗﾍﾞﾙで命令にﾙｰﾌﾟする。
LSL	80286P	LSL reg,mem	セグメント制約をロードする。
LSS	80386	LSS reg,r/m	FAR ﾎﾟｲﾝﾀをSSとﾚｼﾞｽﾀにロードする。
LTR	80286P	LTR mem	ﾀｽｸﾚｼﾞｽﾀ の内容をロードする。
MOV	8086	MOV accum,mem	ﾒﾓﾘの内容をｱｷｭﾑﾚｰﾀに移動させる。
	8086	MOV mem,accum	ｱｷｭﾑﾚｰﾀの内容をﾒﾓﾘに移動させる。
	8086	MOV r/m,immed	ｲﾐﾃﾞｨｴｲﾄ 値を ｵﾍﾟﾗﾝﾄﾞに移動させる。
	8086	MOV r/m,reg	ﾚｼﾞｽﾀを ｵﾍﾟﾗﾝﾄﾞに移動させる。
	8086	MOV r/m,segreg	ｾｸﾞﾒﾝﾄﾚｼﾞｽﾀの内容をｵﾍﾟﾗﾝﾄﾞに移動させる。
	8086	MOV reg,immed	ｲﾐﾃﾞｨｴｲﾄ 値をﾚｼﾞｽﾀに移動させる。
	8086	MOV reg,r/m	ｵﾍﾟﾗﾝﾄﾞ をﾚｼﾞｽﾀに移動させる。
	8086	MOV segreg,r/m	ｵﾍﾟﾗﾝﾄﾞ を ｾｸﾞﾒﾝﾄﾚｼﾞｽﾀに移動させる。
	80386P	MOV reg,Creg	制御ﾚｼﾞｽﾀの内容を汎用ﾚｼﾞｽﾀに移動させる。
	80386P	MOV reg,Dreg	ﾃﾞﾊﾞｯｸﾞﾚｼﾞｽﾀの内容を汎用ﾚｼﾞｽﾀに移動させる。
	80386P	MOV reg,Treg	ﾃｽﾄﾚｼﾞｽﾀ の内容を汎用ﾚｼﾞｽﾀに移動させる。
	80386P	MOVC reg,reg	ﾚｼﾞｽﾀの内容を制御ﾚｼﾞｽﾀに移動させる。
	80386P	MOVD reg,reg	ﾚｼﾞｽﾀの内容をﾃﾞﾊﾞｯｸﾞﾚｼﾞｽﾀに移動させる。
	80386P	MOVT reg,reg	ﾚｼﾞｽﾀの内容を ﾃｽﾄﾚｼﾞｽﾀに移動させる。
MOVS	8086	MOVS dest,src	文字列を移動させる。
MOVSB	8086	MOVSB	BYTE文字列を移動させる。
MOVSD	80386	MOVSD	DWORD 文字列を移動させる。
MOVSW	8086	MOVSW	WORD文字列を移動させる。
MOVSX	80386	MOVSX reg,r/m	符号拡張でﾊﾞｲﾄｵﾍﾟﾗﾝﾄﾞ/ﾜｰﾄﾞｵﾍﾟﾗﾝﾄﾞ をｼﾝｸﾞﾙﾜｰﾄﾞﾚｼﾞｽﾀ/ﾀﾞﾌﾞﾙﾜｰﾄﾞﾚｼﾞｽﾀに移動させる。
MOVZX	80386	MOVZX reg,r/m	ゼロ拡張でﾊﾞｲﾄｵﾍﾟﾗﾝﾄﾞ/ﾜｰﾄﾞｵﾍﾟﾗﾝﾄﾞ をｼﾝｸﾞﾙﾜｰﾄﾞﾚｼﾞｽﾀ/ﾀﾞﾌﾞﾙﾜｰﾄﾞﾚｼﾞｽﾀに移動させる。
MUL	8086	MUL r/m	ｱｷｭﾑﾚｰﾀ を ｵﾍﾟﾗﾝﾄﾞで乗算する。
NEG	8086	NEG r/m	（２の補数）オペランド
NOP	8086	NOP	演算なし
NOT	8086	NOT r/m	（１の補数）オペランド

命　令	必要なプロセッサ	形　　式	説　　　明
OR	8086	OR accum, immed	ｲﾐﾃﾞｨｴｲﾄ 値と ｱｷｭﾑﾚｰﾀの論理和(OR)をとる。
	8086	OR r/m, immed	ｲﾐﾃﾞｨｴｲﾄ 値と ｵﾍﾟﾗﾝﾄﾞの論理和(OR)をとる。
	8086	OR r/m, reg	ﾚｼﾞｽﾀ内容と ｵﾍﾟﾗﾝﾄﾞの論理和(OR)をとる。
	8086	OR reg, r/m	ｵﾍﾟﾗﾝﾄﾞ とﾚｼﾞｽﾀの論理和(OR)をとる。
OUT	8086	OUT DXaccum	DXが指定したﾎﾟｰﾄ に出力する。
	8086	OUT immed, accum	ﾎﾟｰﾄ (8ﾋﾞｯﾄｲﾐﾃﾞｨｴｲﾄ値) に出力する。
OUTS	80186	OUTS DX, mem	文字列をﾎﾟｰﾄ DXに出力する。
OUTSB	80186	OUTSB DX, mem	BYTEをソース文字列からポートDXに出力する。
OUTSD	80386	OUTSD	DWORD をソース文字列からポートDXに出力する。
OUTSW	80186	OUTSW DX, mem	WORDをソース文字列からポートDXに出力する。
POP	8086	POP r/m	ｵﾍﾟﾗﾝﾄﾞ にｽﾀｯｸからポップする。
	8086	POP reg	ﾚｼﾞｽﾀにｽﾀｯｸからポップする。
	8086	POP segreg	ｾｸﾞﾒﾝﾄﾚｼﾞｽﾀ にｽﾀｯｸからポップする。
POPA	80186	POPA	すべての16ビット汎用ﾚｼﾞｽﾀにｽﾀｯｸからポップする。
POPAD	80386	POPAD	すべての32ビット汎用ﾚｼﾞｽﾀにｽﾀｯｸからポップする。
POPF	8086	POPF	ﾌﾗｸﾞﾚｼﾞｽﾀにｽﾀｯｸ からポップする。
POPFD	80386	POPFD	32ﾋﾞｯﾄ EFLAGSﾚｼﾞｽﾀにﾎﾟｯﾌﾟ する。
PUSH	8086	PUSH r/m	16ﾋﾞｯﾄｵﾍﾟﾗﾝﾄﾞの内容をｽﾀｯｸにﾌﾟｯｼｭする。
	8086	PUSH reg	16ﾋﾞｯﾄ ﾚｼﾞｽﾀの内容をｽﾀｯｸにﾌﾟｯｼｭする。
	8086	PUSH segreg	ｾｸﾞﾒﾝﾄ ﾚｼﾞｽﾀの内容をｽﾀｯｸにﾌﾟｯｼｭする。
	80186	PUSH immed	ｲﾐﾃﾞｨｴｲﾄﾃﾞｰﾀをｽﾀｯｸにﾌﾟｯｼｭする。
PUSHA	80186	PUSHA	すべての16ビット汎用ﾚｼﾞｽﾀの内容をｽﾀｯｸにﾌﾟｯｼｭする。
PUSHAD	80386	PUSHAD	すべての32ビット汎用ﾚｼﾞｽﾀの内容をプッシュする。
PUSHF	8086	PUSHF	フラグをｽﾀｯｸにﾌﾟｯｼｭする。
PUSHFD	80386	PUSHFD	32ﾋﾞｯﾄ EFLAGSﾚｼﾞｽﾀの内容をﾌﾟｯｼｭする。

命　令	必要なプロセッサ	形　　式	説　　　　明
RCL	8086 8086 80186	RCL　r/m, 1 RCL　r/m, CL RCL　r/m, immed	ｷｬﾘｰを含めて1ﾋﾞｯﾄ左にﾛｰﾃｰﾄする。 ｷｬﾘｰを含めてCLのｶｳﾝﾄだけ左にﾛｰﾃｰﾄする。 ｷｬﾘｰを含めてｲﾐﾃﾞｨｴｲﾄ値だけ左にﾛｰﾃｰﾄする。
RCR	8086 8086 80186	RCR　r/m, 1 RCR　r/m, CL RCR　r/m, immed	ｷｬﾘｰを含めて1ﾋﾞｯﾄ右にﾛｰﾃｰﾄする。 ｷｬﾘｰを含めてCLのｶｳﾝﾄだけ右にﾛｰﾃｰﾄする。 ｷｬﾘｰを含めてｲﾐﾃﾞｨｴｲﾄ値だけ右にﾛｰﾃｰﾄする。
REP	8086	REP	命令を繰り返す。
REPE	8086	REPE	等しければ命令を繰り返す。
REPNE	8086	REPNE	等しくなければ命令を繰り返す。
REPNZ	8086	REPNZ	ゼロでなければ命令を繰り返す。
REPZ	8086	REPZ	ゼロなら命令を繰り返す。
RET	8086	RET　〔immed〕	ｽﾀｯｸからﾊﾞｲﾄ(ｵﾌﾟｼｮﾝ)をﾎﾟｯﾌﾟして復帰。
ROL	8086 8086 80186	ROL　r/m, 1 ROL　r/m, CL ROL　r/m, immed	左に1ﾋﾞｯﾄﾛｰﾃｰﾄする。 左にCLのｶｳﾝﾄだけﾛｰﾃｰﾄする。 左にｲﾐﾃﾞｨｴｲﾄ値だけﾛｰﾃｰﾄする。
ROR	8086 8086 80186	ROR　r/m, 1 ROR　r/m, CL ROR　r/m, immed	右に1ﾋﾞｯﾄﾛｰﾃｰﾄする。 右にCLのｶｳﾝﾄだけﾛｰﾃｰﾄする。 右にｲﾐﾃﾞｨｴｲﾄ値だけﾛｰﾃｰﾄする。
SAHF	8086	SAHF	AHをﾌﾗｸﾞに記憶する。
SAL	8086 8086 80186	SAL　r/m, 1 SAL　r/m, CL SAL　r/m, immed	左に1ﾋﾞｯﾄ算術ｼﾌﾄする。 左にCLのカウントだけ算術ｼﾌﾄする。 左にｲﾐﾃﾞｨｴｲﾄ値だけ算術ｼﾌﾄする。
SAR	8086 8086 80186	SAR　r/m, 1 SAR　r/m, CL SAR　r/m, immed	右に1ﾋﾞｯﾄ算術ｼﾌﾄする。 右にCLのカウントだけ算術ｼﾌﾄする。 右にｲﾐﾃﾞｨｴｲﾄ値だけ算術ｼﾌﾄする。
SBB	8086 8086 8086 8086	SBB　accum, immed SBB　r/m, immed SBB　r/m, reg SBB　reg, r/m	ﾎﾞﾛｰを含めてｲﾐﾃﾞｨｴｲﾄ値を減ずる。 ﾎﾞﾛｰを含めてｲﾐﾃﾞｨｴｲﾄ値を減ずる。 ﾎﾞﾛｰを含めてﾚｼﾞｽﾀ内容を減ずる。 ﾎﾞﾛｰを含めてｵﾍﾟﾗﾝﾄﾞ内容を減ずる。
SCAS	8086	SCAS dest	文字列をスキャンする。
SCASB	8086	SCASB	文字列をALでｽｷｬﾝし、ﾊﾞｲﾄ比較する。
SCASD	8086	SCASD	文字列をEAXでスキャンし、DWORD比較する。
SCASW	8086	SCASW	文字列をAXでｽｷｬﾝし、ﾜｰﾄﾞ比較する。

命　令	必要なプロセッサ	形　　式	説　　　明
SETA	80386	SETA r/m	より上であればバイトをセットする。
SETAE	80386	SETAE r/m	以上であればバイトをセットする。
SETB	80386	SETB r/m	より下であればバイトをセットする。
SETBE	80386	SETBE r/m	以下であればバイトをセットする。
SETC	80386	SETC r/m	桁上げ時にバイトをセットする。
SETE	80386	SETE r/m	等しければバイトをセットする。
SETG	80386	SETG r/m	より大きければ ﾊﾞｲﾄをセットする。
SETGE	80386	SETGE r/m	以上であればバイトをセットする。
SETL	80386	SETL r/m	未満であればバイトをセットする。
SETLE	80386	SETLE r/m	以下であればバイトをセットする。
SETNA	80386	SETNA r/m	より上でなければバイトをセットする。
SETNAE	80386	SETNAE r/m	以上でなければバイトをセットする。
SETNB	80386	SETNB r/m	未満でなければバイトをセットする。
SETNBE	80386	SETNBE r/m	以下でなければバイトをセットする。
SETNC	80386	SETNC r/m	桁上げのない時にﾊﾞｲﾄをｾｯﾄする。
SETNE	80386	SETNE r/m	等しくなければバイトをセットする。
SETNG	80386	SETNG r/m	より大きくなければ ﾊﾞｲﾄを ｾｯﾄする。
SETNGE	80386	SETNGE r/m	以下であればバイトをセットする。
SETNL	80386	SETNL r/m	より小さくなければ ﾊﾞｲﾄを ｾｯﾄする。
SETNLE	80386	SETNLE r/m	以下でなければバイトをセットする。
SETNO	80386	SETNO r/m	ｵｰﾊﾞｰﾌﾛｰ しない時に ﾊﾞｲﾄを ｾｯﾄする。
SETNP	80386	SETNP r/m	ﾊﾟﾘﾃｨﾌﾗｸﾞ=0 であれば ﾊﾞｲﾄを ｾｯﾄする。
SETNS	80386	SETNS r/m	符号のない時に ﾊﾞｲﾄを ｾｯﾄする。
SETNZ	80386	SETNZ r/m	ゼロでなければ ﾊﾞｲﾄを ｾｯﾄする。
SETO	80386	SETO r/m	ｵｰﾊﾞｰﾌﾛｰ 時に ﾊﾞｲﾄを ｾｯﾄする。
SETP	80386	SETP r/m	ﾊﾟﾘﾃｨﾌﾗｸﾞ が立った時にバイトをセットする。
SETPE	80386	SETPE r/m	偶数ﾊﾟﾘﾃｨならバイトをセットする。
SETPO	80386	SETPO r/m	奇数ﾊﾟﾘﾃｨならバイトをセットする。
SETS	80386	SETS r/m	符号ﾌﾗｸﾞ=1 なら ﾊﾞｲﾄを ｾｯﾄする。
SETZ	80386	SETZ r/m	ゼロならバイトをセットする。
SGDT	80286P	SGDT mem	GDT ﾚｼﾞｽﾀをメモリ (24ﾋﾞｯﾄﾍﾞｰｽ) に記憶する。
SGDTE	80386P	SGDTE　mem	GDT ﾚｼﾞｽﾀをメモリ (32ﾋﾞｯﾄﾍﾞｰｽ) に記憶する。

命　令	必要なプロセッサ	形　式	説　　明
SHL	8086 8086 80186	SHL r/m,1 SHL r/m,CL SHL r/m,immed	左に１ビットシフトする。 左にCLビットだけシフトする。 左にｲﾐﾃｨｴｲﾄ 値だけシフトする。
SHLD	80386 80386	SHLD r/m,reg,cl SHLD r/m,reg,immed	CLﾋﾞｯﾄ だけﾚｼﾞｽﾀから左にr/m へシフトする。 ｲﾐﾃｨｴｲﾄ 値ﾋﾞｯﾄｶｳﾝﾄ だけﾚｼﾞｽﾀから左にr/m へシフトする。
SHR	8086 8086 80186	SHR r/m,1 SHR r/m,CL SHR r/m,reg,immed	右に１ビットシフトする。 右にCLだけビットシフトする。 右にｲﾐﾃｨｴｲﾄ 値だけシフトする。
SHRD	80386 80386	SHRD r/m,reg,cl SHRD r/m,reg,immed	CLﾋﾞｯﾄ だけﾚｼﾞｽﾀから右にr/m へシフトする。 ｲﾐﾃｨｴｲﾄ 値ﾋﾞｯﾄｶｳﾝﾄ だけﾚｼﾞｽﾀから右r/m へシトする。
SIDT	80286P	SIDT mem	IDT ﾚｼﾞｽﾀをメモリ(24ﾋﾞｯﾄﾍﾞｰｽ)に記憶する。
SIDTE	80386P	SIDTE mem	IDT ﾚｼﾞｽﾀをメモリ(32ﾋﾞｯﾄﾍﾞｰｽ)に記憶する。
SLDT	80286P	SLDT mem	ﾛｰｶﾙﾃﾞｨｽｸﾘﾌﾟﾀﾃｰﾌﾞﾙﾚｼﾞｽﾀを記憶する。
SMSW	80286P	SMSW mem	ﾏｼﾝｽﾃｰﾀｽﾜｰﾄﾞ を記憶する。
STC	8086	STC	キャリーフラグをセットする。
STD	8086	STD	ﾃﾞｨﾚｸｼｮﾝﾌﾗｸﾞを セットする。
STI	8086	STI	割込みフラグをセットする。
STOS	8086	STOS dest	文字列をストアする。
STOSB	8086	STOSB	BYTEを宛先文字列にALからｽﾄｱ する。
STOSD	8086	STOSD	DWORD を宛先文字列にEAXからストアする。
STOSW	8086	STOSW	WORDを宛先文字列にAXからｽﾄｱ する。
STR	80286P	STR mem	ﾀｽｸﾚｼﾞｽﾀ の内容をストアする。
SUB	8086 8086 8086 8086	SUB accum,immed SUB r/m,immed SUB r/m,reg SUB reg,r/m	ｲﾐﾃｨｴｲﾄ値をｱｷｭﾑﾚｰﾀから減算する。 ｲﾐﾃｨｴｲﾄ 値を ｵﾍﾟﾗﾝﾄﾞから減算する。 ﾚｼﾞｽﾀの内容を ｵﾍﾟﾗﾝﾄﾞから減算する。 ｵﾍﾟﾗﾝﾄﾞ をﾚｼﾞｽﾀの内容から減算する。
TEST	8086 8086 8086 8086	TEST accum,immed TEST r/m,immed TEST r/m,reg TEST reg,r/m	ｱｷｭﾑﾚｰﾀ をｲﾐﾃｨｴｲﾄﾋﾞｯﾄで ﾃｽﾄする。 ｵﾍﾟﾗﾝﾄﾞ をｲﾐﾃｨｴｲﾄﾋﾞｯﾄで ﾃｽﾄする。 ｵﾍﾟﾗﾝﾄﾞ を ﾚｼﾞｽﾀﾋﾞｯﾄで ﾃｽﾄする。 ﾚｼﾞｽﾀをｵﾍﾟﾗﾝﾄﾞﾋﾞｯﾄで ﾃｽﾄする。
VERR	80286P	VERR mem	読取りｱｸｾｽをベリファイする。

命　令	必要なプロセッサ	形　　式	説　　　　明
VERW	80286P	VERW mem	書込みｱｸｾｽをベリファイする。
WAIT	8086	WAIT	待つ。
XCHG	8086	XCHG accum, reg	ｱｷｭﾑﾚｰﾀ の内容をﾚｼﾞｽﾀの内容と交換する。
	8086	XCHG r/m, reg	ｵﾍﾟﾗﾝﾄﾞ をﾚｼﾞｽﾀの内容と交換する。
	8086	XCHG reg, accum	ﾚｼﾞｽﾀの内容を ｱｷｭﾑﾚｰﾀの内容と交換する。
	8086	XCHG reg, r/m	ﾚｼﾞｽﾀの内容を ｵﾍﾟﾗﾝﾄﾞと交換する。
XLAT	8086	XLAT mem	トランスレートする。
XOR	8086	XOR accum, immed	ｲﾐﾃﾞｨｴｲﾄ値とｱｷｭﾑﾚｰﾀとを XORする。
	8086	XOR r/m, immed	ｲﾐﾃﾞｨｴｲﾄ 値と ｵﾍﾟﾗﾝﾄﾞとを XORする。
	8086	XOR r/m, reg	ﾚｼﾞｽﾀの内容を ｵﾍﾟﾗﾝﾄﾞとを XORする。
	8086	XOR reg, r/m	ｵﾍﾟﾗﾝﾄﾞ とﾚｼﾞｽﾀの内容とを XORする。

表D－2　80287 命令セット

命　令	必要なプロセッサ	形　　式	説　　　明
F2XMI	8087	F2XM1	2^x-1を計算する。
FABS	8087	FABS	スタックのトップの絶対値をとる。
FADD	8087	FADD	ST(1)をSTに加えてポップする。
	8087	FADD mem	メモリ中の実数をSTに加える。
	8087	FADD ST, ST(i)	ST(i)をSTに加える。
	8087	FADD ST(i), ST	STをST(i)に加えてポップする。
FADDP	8087	FADD ST(i), ST	STをST(i)に加えてｽﾀｯｸをﾎﾟｯﾌﾟする。
FBLD	8087	FBLD mem	10ﾊﾞｲﾄのBCD値をｽﾀｯｸにロードする。
FBSTP	8087	FBSTP mem	10ﾊﾞｲﾄのBCD値を記憶してﾎﾟｯﾌﾟする。
FCHS	8087	FCHS	スタックのトップの符号を変える。
FCLEX	8087	FCLEX	WAITのあと例外をクリヤする。
FCOM	8087	FCOMP	STをST(1)と比較する。
	8087	FCOMP ST	STをメモリの実数と比較する。
	8087	FCOMP ST(i)	STをST(i)と比較する。
FCOMP	8087	FCOM	STをST(i) と比較してｽﾀｯｸをﾎﾟｯﾌﾟ する。
	8087	FCOM ST	STをﾒﾓﾘ と比較して、ポップする。
	8087	FCOM ST(i)	実数をｽﾀｯｸと比較して、ｽﾀｯｸをﾎﾟｯﾌﾟする。
FCOMPP	8087	FCOMPP	STをST(i)と比較してスタックを２回ポップする。
FDECSTP	8087	FDECSTP	浮動小数点ｽﾀｯｸﾎﾟｲﾝﾀから１を引く。
FDISI	8087のみ	FDISI	WAITのあと割込みを不可にする。（ターゲットが287 の場合にはコードを生成しない）
FDIV	8087	FDIV	ST(1) をSTで除算して、ﾎﾟｯﾌﾟする。
	8087	FDIV mem	ST をメモリの実数で除算する。
	8087	FDIV ST, ST(i)	ST をST(i)で除算する。
	8087	FDIV ST(i), ST	ST(i) をSTで除算する。
FDIVP	8087	FDIVP ST(i), ST	ST(i) をSTで除算し、スタックをポップする。
FDIVR	8087	FDIVR	STをST(i) で除算し、結果をST(i)に残してからポップする。
	8087	FDIVR mem	ﾒﾓﾘ をSTで除算し、結果をSTに残す。
	8087	FDIVR ST, ST(i)	ST(i) をSTで除算し、結果をSTに残す。
	8087	FDIVR ST(i), ST	STをST(i) で除算し、結果をST(i)に残す。

命 令	必要なプロセッサ	形 式	説 明
FDIVRP	8087	FDIVRP ST(i),ST	STをST(i) で除算し、結果をST (i) に残してからポップする。
FENI	8087のみ	FENI	WAITのあと割込みを可能にする。(ターゲットが287 の場合にはコードを生成しない。
FFREE	8087 8087 8087	FFREE FFREE ST FFREE ST(i)	STをフリーとしてマークする。 STをフリーとしてマークする。 ST(i)をフリーとしてマークする。
FIADD	8087	FIADD mem	STに2バイトか4バイトの整数を加える。
FICOM	8087	FICOM mem	STを2バイトか4バイトの整数と比較する。
FICOMP	8087	FICOMP mem	STを2バイトか4バイトの整数と比較してポップする。
FIDIV	8087	FIDIV mem	STを2バイトか4バイトの整数で除算する。
FIDIVR	8087	FIDIVR mem	2バイトか4バイトの整数をSTで除算し、結果をSTに残す。
FILD	8087	FILD mem	2バイト、4バイト、8バイトの整数を浮動小数点スタックにプッシュする。
FIMUL	8087	FIMUL mem	2バイトか、4バイトの整数乗算
FINCSTP	8087	FINCSTP	浮動小数点スタックポインタに1を加える。
FINIT	8087	FINIT	WAITのあとプロセッサ を初期化する。
FIST	8087	FIST mem	STを2バイトか4バイトの整数としてmem に記憶する。
FISTP	8087	FISTP mem	STを2バイトまたは8バイトの整数として記憶してポップする。
FISUB	8087	FISUB mem	2バイトまたは4バイトの整数をSTから減算する。
FISUBR	8087	FISUBR mem	STを2バイトまたは4バイトの整数から減算し、結果をSTに残す。
FLD	8087	FLD mem	4バイト、8バイト、または10バイト の実数を浮動小数点スタックにプッシュする。
FLDCW	8087	FLDCW mem	コプロセッサ制御ワードにロードする。
FLDENV	8087	FLDENV mem	コプロセッサ環境(14バイト) にロードする。
FLDI	8087	FLDI	1.0 をプッシュする。
FLDL2E	8087	FLDL2E	$\log_2 e$を浮動小数点スタックにプッシュする。
FLDL2T	8087	FLDL2T	$\log_2 10$を浮動小数点スタックにプッシュする。

命　令	必要なプロセッサ	形　　式	説　　　　明
FLDLG2	8087	FLDLG2	$log_{10}2$を浮動小数点スタックにプッシュする。
FLDLN2	8087	FLDLN2	$log_e 2$ を浮動小数点スタックにプッシュする。
FLDPI	8087	FLDLPI	πをｽﾀｯｸのトップにプッシュする。
FLDZ	8087	FLDZ	+0.0を浮動小数点ｽﾀｯｸにﾛｰﾄﾞ する。
FMUL	8087 8087 8087 8087	FMUL FMUL mem FMULST, ST(i) FMULST(i), ST	ST(1)にSTを乗算し、ポップする。 STにメモリの実数を乗算する。 STにST(i) を乗算する。 ST(i) にSTを乗算する。
EMULP	8087	FMULT ST(i),ST	ST(i) にSTを乗算し、ポップする。
FNCLEX	8087	FNCLEX	WAITなしで例外をクリヤする。
FNDISI	8087のみ	FNDISI	WAITなしで割込みを不可にする。（ターゲットが287 の場合にはコードを生成しない）
FNENI	8087のみ	FNENI	WAITなしで割込みを有効にする。（ターゲットが287 の場合、コードを生成しない）
FNINIT	8087	FNINIT	WAITなしでプロセッサを初期化する。
FNOP FNSAVE	8087 8087	FNOP FNSAVE mem	演算しない。 ｺﾌﾟﾛｾｯｻの状態(94ﾊﾞｲﾄ) を保存する。WAITなし
FNSTCW	8087	FNSTCW mem	WAITなしでｺﾌﾟﾛｾｯｻ制御ワードをストアする。
FNSTENV	8087	FNSTENV mem	WAITなしでコプロセッサ環境（14バイト）をストアする。
FNSTSW	8087 80287	FNSTSW mem FNSTSW AX	WAITなしでｺﾌﾟﾛｾｯｻのステータスワードをストアする。 WAITなしでAXの中の80287 ステータスをストアする。
FPATAN	8087	FPATAN	arctan(ST(1)/ST)を計算し、結果を1 度ポップしたあとSTに残す。
FPREM	8087	FPREM	STmodST(1)の部分ﾓｼﾞｭﾗｽ を計算。
FPTAN	8087	FPTAN	tan(ST)を計算する。
FRNDINT	8087	FRNDINT	STを整数にまるめる。
FRSTOR	8087	FRSTOR mem	ｺﾌﾟﾛｾｯｻの ｽﾃｰﾀｽ(64ﾊﾞｲﾄ)を復元する。
FSAVE	8087	FSAVE mem	WAITのあとの8087ステータス(94ﾊﾞｲﾄ)を退避する。

命　令	必要なプロセッサ	形　　式	説　　　明
FSCALE	8087	FSCALE	ST(1) をSTの指数に加える。
FSETPM	80287	FSETPM	80287を386ﾈｲﾃｨﾌﾞﾓｰﾄﾞ にｾｯﾄ する。
FSQRT	8087	FSQRT	STの平方根を計算し、結果をSTに残す。
FST	8087 8087	FST ST(i) FST mem	ST(i) にSTをストアする。 STを4ﾊﾞｲﾄか8ﾊﾞｲﾄの実数としてmem にストアする。
FSTCW	8087	FSTCW mem	WAITでｺﾌﾟﾛｾｯｻ制御ﾜｰﾄﾞ をｽﾄｱ する。
FSTENV	8087	FSTENV mem	WAITのあとｺﾌﾟﾛｾｯｻ環境をｽﾄｱ する。
FSTP	8087	FSTP mem	STを4ﾊﾞｲﾄ,8ﾊﾞｲﾄ または10ﾊﾞｲﾄ の実数としてストアし、ポップする。
FSTSW	8087 80287	FSTSW mem FSTSW AX	WAITのあとｺﾌﾟﾛｾｯｻｽﾃｰﾀｽﾜｰﾄﾞをストアする。 WAITのあとAXに80287 ｽﾃｰﾀｽﾜｰﾄﾞをストアする。
FSUB	8087 8087 8087 8087	FSUB mem FSUB mem FSUB ST,ST(i) FSUB ST(i), ST	ST(1) からSTを減算し、ﾎﾟｯﾌﾟ する。 ﾒﾓﾘ の実数をSTから減算する。 ST(i) をSTから減算する。 STをST(i) から減算する。
FSUPBP	8087	FSUBP ST(i), ST	STをST(i) から減算し、ﾎﾟｯﾌﾟ する。
FSUBR	8087 8087 8087 8087	FSUBR FSUBR mem FSUBR ST,ST(i) FSUBR ST(i), ST	ST(1) をSTから減算し、結果をST(1) に残してからポップする。 STをメモリ実数から減算し、結果をSTに残す。 STををST(i) から減算し、結果をSTに残す。 ST(i) をSTから減算し、結果をST(i) に残す。
FSUBRP	8087	FSUBRP	ST(i) をSTから減算し、結果をST(i) に残してからポップする。
FTST	8087	FTST	STを0.0 と比較する。
FWAIT	8087	FWAIT	最後のコプロセッサ演算が完了するのを待つ。
FXAM	8087	FXAM	STを検査し、コプロセッサ条件コードをセットする。
FXCH	8087 8087	FXCH FXCH ST(i)	STとST(1) を交換する。 STとST(i) を交換する。
FXTRACT	8087	FXTRACT	STを指数と有意部分に分解する。

命　令	必要なプロセッサ	形　　式	説　　　　明
FYL2X	8087	FYL2X	$Y\ \log_2 x$ を計算する。
FYL2P1	8087	FYL2P1	$Y\ \log_2(x+1)$を計算する。

付録E ディレクティブの概略

表E－1　アセンブルディレクティブ

ディレクティブ名	文の形式	説明
.186	.186	80186 命令のアセンブルを可能にする。
.286c .286	.286c .286	80286 非保護命令のアセンブルを可能にする。
.286p	.286p	すべての80286 命令のアセンブルを可能にする。
.287	.287	80287 命令のアセンブルを可能にする。
.386c .386	.386c .386	80386 非保護命令のアセンブルを可能にする。これはデフォルトモードである。
.386p	.386p	すべての80386 命令のアセンブルを可能にする。
.387	.387	80387 命令のアセンブルを可能にする。
.8086	.8086	8086命令のアセンブルを可能にする。
.8087	.8087	8087命令のアセンブルを可能にする。

ディレクティブ名	文の形式	説　　明
	neme=expression	expressionの値を持つシンボルname名を作成する。シンボルは定数、変数またはラベルである。
ASSUME	ASSUME registername: segname	ﾌﾟﾛｸﾞﾗﾑ 実行時 registername がsegname と呼ぶｾｸﾞﾒﾝﾄ またはｸﾞﾙｰﾌﾟを指すことをｱｾﾝﾌﾞﾗ に示す。segname がNOTHING の場合、ﾚｼﾞｽﾀは未定義値をもつと仮定される。
COMMENT	COMMENT delim text delim	delim 間のすべてのtextをｺﾒﾝﾄとしてｱｾﾝﾌﾞﾗ に処理させる。Textは改行もできるので複数行のｺﾒﾝﾄも作成できる。
.DB	{name} DB value,...	value ごとに1ﾊﾞｲﾄ のﾒﾓﾘ の割当てと初期化を行う。name(ｵﾌﾟｼｮﾝ) がある場合、BYTE型の変数が指定名で作成される。
DW	{name} DW value,...	value ごとに1ﾜｰﾄﾞ(2ﾊﾞｲﾄ) のﾒﾓﾘ の割当てと初期化を行う。name(ｵﾌﾟｼｮﾝ)がある場合、WORD型の変数が指定名で作成される。
DD	{name} DD value,...	value ごとに1ﾀﾞﾌﾞﾙﾜｰﾄﾞ(4ﾊﾞｲﾄ) のﾒﾓﾘの割当と初期化を行う。name(ｵﾌﾟｼｮﾝ)がある場合、DWORD 型の変数が指定名で作成される。
DP DF	{name} DP value,... {name} DF value,...	value ごとに3ﾜｰﾄﾞ(6ﾊﾞｲﾄ)のﾒﾓﾘ の割当てと初期化を行う。name(ｵﾌﾟｼｮﾝ)がある場合、PWORD 型の変数が指定名で作成される。

ディレクティブ名	文の形式	説　　明
DQ	{name} DQ value,...	value ごとにｸﾜｯﾄﾞﾜｰﾄﾞ(8ﾊﾞｲﾄ) のﾒﾓﾘの割当てと初期化を行う。name(ｵﾌﾟｼｮﾝ) がある場合、QWORD 型の変数が指定名で作成される。
DT	{name} DT value,...	value ごとに10ﾊﾞｲﾄ のﾒﾓﾘ の割当てと初期化を行う。name(ｵﾌﾟｼｮﾝ) がある場合、TBYTE 型の変数が指定名で作成される。
ELSE	ELSE	条件付きﾌﾞﾛｯｸの真部分の終りと偽部分の始めとを示す。
END	END {entry }	ｱｾﾝﾌﾞﾙ 中のﾌﾟﾛｸﾞﾗﾑﾓｼﾞｭｰﾙの終りを表示。そのﾌﾟﾛｸﾞﾗﾑ のｴﾝﾄﾘﾎﾟｲﾝﾄをentry とし確定する(ｵﾌﾟｼｮﾝ)。
ENDIF	ENDIF	条件付きﾌﾞﾛｯｸの終りを通知する。
ENDM	ENDM	ﾏｸﾛ またはリピートﾌﾞﾛｯｸの終りを通知する。
ENDP	name ENDP	nameに対してﾌﾟﾛｼｼﾞｬ 定義の終りを通知する。
ENDS	name ENDS	nameという名のｾｸﾞﾒﾝﾄ をｸﾛｰｽﾞするか、nameの構造体定義を終了させる。
EQU	name EQU expression	expressionの値をnameに割当てる。expressionはｱﾄﾞﾚｽ式、ｱｾﾝﾌﾞﾗｷｰﾜｰﾄﾞ、任意のテキスト文字列または定数値である。
.ERR	.ERR	ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRB	.ERRB <string>	stringが空白の場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。

ディレクティブ名	文の形式	説　　明
.ERRDEF	.ERRDEF <name>	ｱｾﾝﾌﾞﾗ のｼﾝﾎﾞﾙﾃｰﾌﾞﾙ中にnameというｼﾝﾎﾞﾙがあり、そのｼﾝﾎﾞﾙが現ｿｰｽﾗｲﾝの前に定義されたものである場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRDIF	.ERRDIF <str1>,<STR2>	文字列str1とstr2が異なる場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRE	.ERRE expression	与えられたexpressionが偽である(0と評価された）場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRIDN	.ERRIDN <str1>,<str2>	文字列str1とstr2が同じ場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRNB	.ERRNB <string>	stringが空白でない場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRNDEF	.ERRNDEF <name>	ｱｾﾝﾌﾞﾗ のｼﾝﾎﾞﾙﾃｰﾌﾞﾙ中にnameと名付けられたｼﾝﾎﾞﾙがない場合、または現ｿｰｽﾗｲﾝの後にnameが定義されている場合、ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
.ERRNZ	.ERRNZ expression	与えられたexpressionが真であれば（０以外の値に評価された）ｱｾﾝﾌﾞﾗｴﾗｰを強制的に発生させる。
EVEN	EVEN	必要なら、NOP 命令(90H) を生成することによって、ｶﾚﾝﾄｵｰﾌﾟﾝｾｸﾞﾒﾝﾄ のﾛｹｰｼｮﾝｶｳﾝﾀｰ を、偶数値に調整する。
EXITM	EXITM	現ﾏｸﾛ 展開またはリピートﾌﾞﾛｯｸを直ちに強制的に終了させる。
EXTRN	EXTRN name:type,...	nameという名のｼﾝﾎﾞﾙは現ﾓｼﾞｭｰﾙ に対し外部ｼﾝﾎﾞﾙであり、type型であることをｱｾﾝﾌﾞﾗ に知らせる。

ディレクティブ名	文の形式	説明
GROUP	name GROUP segname,...	ﾃﾞｨﾚｸﾃｨﾌﾞ のあとの１つまたは複数のｾｸﾞをメンバーとするグループnameを定義する。
IF	IF expression	expressionが真の（０以外の値に評価された）場合、条件付きﾌﾞﾛｯｸの真の部分に命令をｱｾﾝﾌﾞﾙ する。
IFB	IFB <string>	stringが空白の場合、条件付ﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。
IFDEF	IFDEF name	ｱｾﾝﾌﾞﾗ のｼﾝﾎﾞﾙﾃｰﾌﾞﾙ中にnameという名のｼﾝﾎﾞﾙがあり、そのｼﾝﾎﾞﾙが現ｿｰｽﾗｲﾝの前に定義されていた場合、条件付きﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する
IFDIF	IFDIF <str1>,<str2>	str1とstr2が異なる場合、条件付きﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。
IFE	IFE expression	expressionが偽の(0と評価された)場合、条件付きﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。
IFIDN	IFIDN <str1>,<str2>	str1とstr2が同じ場合、条件付きﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。
IFNB	IFNB <string>	stringが空白でない場合、条件付ﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。
IFNDEF	IFNDEF name	ｱｾﾝﾌﾞﾗ のｼﾝﾎﾞﾙﾃｰﾌﾞﾙ中にnameという名のｼﾝﾎﾞﾙがない場合、または、nameが現ｿｰｽﾗｲﾝのあとに定義されている場合、条件付きﾌﾞﾛｯｸの真部分の命令をｱｾﾝﾌﾞﾙ する。

ディレクティブ名	文の形式	説　　　明
INCLUDE	INCLUDE filename	filenameによって指定されたソースファイル 中のテキストを、アセンブル 中のファイルに挿入する。
IRP	IRP fparam, <aparam,...>	実パラメータ aparamごとに１回繰返すリピートブロックを開始する。この時、リピートブロックテキスト中の形式パラメータ fparamは実パラメータ 値に置き換えられる。
IRPC	IRPC fparam,string	string中の文字ごとに１回繰返すリピートブロックを開始する。この時、リピートブロックテキスト中のパラメータfparamはその文字に置き換えられる。
LABEL	name LABEL type	カレントセグメント のロケーションカウンタと等しい値を持ったtype型の変数または命令ラベル を新しく作成する。
.LALL	.LALL	マクロ 展開中のすべての文をアセンブラリスティングファイル 中にリスト させる。
.LFCOND	.LFCOND	偽の条件付きブロック中のテキストを、アセンブル されていなくても、アセンブラリスティングファイル にコピー する。
.LIST	.LIST	アセンブラリスティングファイル 中のプログラム 文のリスティングを可能にする。
.LISTI	.LIST	INCLUDE ディレクティブ でインクルードされたファイルからソースラインのリスティングをできるようにする。
LOCAL	LOCAL dname,...	マクロ 内で使用するため単数または複数のダミー 名を作成する。ダミー 名は、アセンブラの生成する特有な名前（形式：？？××××。×はマクロ 展開時の16進数）によって置き換えられる。

ディレクティブ名	文の形式	説　　明
MACRO	name MACROP fparam,...	形式ﾊﾟﾗﾒｰﾀ 名fparamを持つ、ﾏｸﾛ nameのﾏｸﾛ 定義ﾌﾞﾛｯｸを開始する。
NAME	NAME modulename	ｱｾﾝﾌﾞﾙ 中のﾓｼﾞｭｰﾙ 名をmodule-nameの先頭132 文字にする。
ORG	ORG expression	ｶﾚﾝﾄｵｰﾌﾟﾝｾｸﾞﾒﾝﾄ のﾛｹｰｼｮﾝｶｳﾝﾀ値をexpressionに設定する。
%OUT	%OUT text	ｱｾﾝﾌﾞﾙ のﾊﾟｽ1とﾊﾟｽ2 の両方でﾕｰｻﾞ 端末にtextを出力する。
%OUT1	%OUT1 text	ｱｾﾝﾌﾞﾘ のﾊﾟｽ1 でtextをﾕｰｻﾞ 端末に出力する。
%OUT2	%OUT2 text	ｱｾﾝﾌﾞﾘ のﾊﾟｽ2 でtextをﾕｰｻﾞ 端末に出力する。
PAGE	PAGE(length),(width)	ﾘｽﾃｨﾝｸﾞﾌｧｲﾙのﾍﾟｰｼﾞ 長とﾍﾟｰｼﾞ 幅をそれぞれlengthとwidth に設定する。
	PAGE　+	ﾘｽﾃｨﾝｸﾞﾌｧｲﾙﾍﾟｰｼﾞ の見出し行の項目番号をｲﾝｸﾘﾒﾝﾄ し、ﾍﾟｰｼﾞﾌﾞﾚｰｸ を生成する。
	PAGE	ｱｾﾝﾌﾞﾗﾘｽﾃｨﾝｸﾞﾌｧｲﾙ 内にﾍﾟｰｼﾞﾌﾞﾚｰｸ を生成する。
PROC	name PROC type	type型のnameと呼ばれるﾌﾟﾛｼｼﾞｬ の定義を開始する。
.PROT	.PROT	80386 または80286 の保護命令のアセンブルを可能にする。
PUBLIC	PUBLIC name,...	変数、命令ﾗﾍﾞﾙ 、または絶対ｼﾝﾎﾞﾙのﾘｽﾄ をﾌﾟﾛｸﾞﾗﾑ 中のすべてのﾓｼﾞｭｰﾙ に使えるようにする。

ディレクティブ名	文の形式	説　　明
	PURGE macroname	なにもしない。
.RADIX	.RADIX expression	入力ファイル中の数字のデフォルト 基数をexpressionに設定する。カレントデフォルト基数に関係なくexpressionの中の数字は常に基数10で求められる。
RECORD	name RECORD fname:width {=expr } ..	nameと呼ばれるレコードをfname という名のビットフィールドで定義する。フィールド 幅は、（単位:ビット）は定数値width によって示され、=expr のある場合は、そのフィールド のデフォルト 値を定義する。
REPT	REPT expression	expression回数繰返すブロックの始め。
.SALL	.SALL	アセンブラリスティングファイル 中のマクロ 展開のリスティングを停止する。
SEGMENT	name SEGMENT {align} {combine } {useatr} {access} {'class' }	align, combine, useatr, access, class の属性を持つnameと呼ばれるプログラムセグメントをオープンする。
.SFCOND	.SFCOND	以降の偽条件付きブロックをアセンブラリスティングファイル 中にリスティングすることを停止する。
STRUC	name STRUC	構造体定義の開始を通知する。
SUBTTL	SUBTTL {text}	リスティングファイルのページ 見出しに使うサブタイトルをtextに設定する。
.TFCOND	.TFCOND	" リスト 偽条件付きブロック"フラグの現状値を交互に変更する。
TITLE	TITLE text	リスティングファイルのページ 見出しに使うタイトルをtextに設定する。

ディレクティブ名	文の形式	説　　明
.XALL	.XALL	マクロ 展開でオブジェクトコード を生成する文だけをアセンブラリスティングファイル の中にリストさせる。
.XLIST	.XLIST	以降のソースラインをアセンブラリスティングファイル 中にリスティングすることを停止する。
.XLISTI	.XLISTI	INCLUDE ディレクティブ を使ってインクルードされたファイルからソースラインをリスティングすることを不可にする。

付録 F　レジスタ名

表F－1　80386 レジスタ

レジスタ名	80386	説　　明
EAX	のみ有	32ビット汎用レジスタ
EBX	のみ有	32ビット汎用レジスタ
ECX	のみ有	32ビット汎用レジスタ
EDX	のみ有	32ビット汎用レジスタ
ESI	のみ有	32ビット汎用レジスタ
EDI	のみ有	32ビット汎用レジスタ
EBP	のみ有	32ビット汎用レジスタ
ESP	のみ有	32ビットスタックポインタ
AX	以外も有	16ﾋﾞｯﾄ 汎用ﾚｼﾞｽﾀ, EAX の下位ﾜｰﾄﾞ
BX	以外も有	16ﾋﾞｯﾄ 汎用ﾚｼﾞｽﾀ, EBX の下位ﾜｰﾄﾞ
CX	以外も有	16ﾋﾞｯﾄ 汎用ﾚｼﾞｽﾀ, ECX の下位ﾜｰﾄﾞ
DX	以外も有	16ﾋﾞｯﾄ 汎用ﾚｼﾞｽﾀ, EDX の下位ﾜｰﾄﾞ
SI	以外も有	16ﾋﾞｯﾄｲﾝﾃﾞｸｽﾚｼﾞｽﾀ, ESI の下位ﾜｰﾄﾞ
DI	以外も有	16ﾋﾞｯﾄｲﾝﾃﾞｸｽﾚｼﾞｽﾀ, EDI の下位ﾜｰﾄﾞ
BP	以外も有	16ﾋﾞｯﾄﾍﾞｰｽﾚｼﾞｽﾀ, EBP の下位ﾜｰﾄﾞ
SP	以外も有	16ﾋﾞｯﾄｽﾀｯｸﾎﾟｲﾝﾀ , ESP の下位ﾜｰﾄﾞ
AH	以外も有	AXの上位バイト
AL	以外も有	AXの下位バイト
BH	以外も有	BXの上位バイト
BL	以外も有	BXの下位バイト
CH	以外も有	CXの上位バイト
CL	以外も有	CXの下位バイト

レジスタ名	80386	説　　　明
DH	以外も有	DXの上位バイト
DL	以外も有	DXの下位バイト
CS	以外も有	コードセグメントセレクタレジスタ
DS	以外も有	データセグメントセレクタレジスタ
ES	以外も有	特殊データセグメントセレクタレジスタ
FS	のみ有	特殊データセグメントセレクタレジスタ
GS	のみ有	特殊データセグメントセレクタレジスタ
SS	以外も有	スタックセグメントセレクタレジスタ
CR0	のみ有	マシン 制御レジスタ(80286 マシンステータスワードを含む)
CR2	のみ有	ページ フォールトリニヤアドレスレジスタ
CR3	のみ有	ページディレクトリベースレジスタ
DR 0	のみ有	リニアブレークポイント アドレス0レジスタ
DR 1	のみ有	リニアブレークポイント アドレス1レジスタ
DR 2	のみ有	リニアブレークポイント アドレス2レジスタ
DR 3	のみ有	リニアブレークポイント アドレス3レジスタ
DR 6	のみ有	ブレークポイントステータス レジスタ
DR 7	のみ有	ブレークポイント制御レジスタ
TR 6	のみ有	テスト制御レジスタ
TR 7	のみ有	テストステータス レジスタ
ST(0) -	以外も有	8087浮動小数点スタックレジスタ
ST(7)	以外も有	

付録G データタイプ

386｜ASM は、80386 および80287 の全範囲のデータタイプをサポートします。データタイプとその値の範囲を表G－1に示します。

386｜ASM が行う整数値の内部演算は、すべて33ビットの符号付表現を用います。これは、式評価中の中間結果について、可能な数値範囲を拡大します。式評価の最終結果は、強制的に適切なデータタイプを与えられ、その結果が許容範囲を超えると、整数オーバーフローエラーが報告されます。

表G－1　データタイプと範囲

ﾃﾞｰﾀﾀｲﾌﾟ	80386	ﾃﾞｨﾚｸﾃｨﾌﾞ	ﾋﾞｯﾄ	説　明	範囲（概算）
BYTE	以外も有	DB	8	80386ﾊﾞｲﾄ	-128から127(符号付き整数) 0 から255(符号無し整数)
WORD	以外も有	DW	16	80386ﾜｰﾄﾞ 80287ﾜｰﾄﾞ整数	-32,768 から32,767（符号付き整数） 0 から65535(符号無し整数)
DWORD	以外も有	DD	32	80386ﾀﾞﾌﾞﾙﾜｰﾄﾞ 80287ｼｮｰﾄ整数 80287ｼｮｰﾄ実数	-2^{31}から$2^{31}-1$(符号付き整数) 0 から$2^{32}-1$(符号無し整数) $-3.4x10^{38}$から$-1.2x10^{-38}$ 0 $1.2x10^{-38}$ から$3.4x10^{38}$ （以上3行）実数
PWORD FWORD	のみ有	DP DF	48	80386ﾎﾟｲﾝﾀﾜｰﾄﾞ または 16ﾋﾞｯﾄｾｸﾞﾒﾝﾄｾﾚｸﾀ＋ﾋﾞｯﾄ 符号なしｵﾌｾｯﾄ	-2^{47}から$2^{47}-1$(符号付き整数) 0 から$2^{48}-1$(符号無し整数)
QWORD	以外も有	DQ	64	80287ﾛﾝｸﾞ整数 80287ﾛﾝｸﾞ実数	-2^{63}から$2^{63}-1$(符号付き整数) 0 から$2^{64}-1$(符号無し整数) $-1.7x10^{308}$から $-2.3x10^{-308}$ 0 $2.3x10^{-308}$から $1.7x10^{308}$ （以上5行）実数
TBYTE	以外も有	DT	80	80287ﾊﾟｯｸ10進数 80287 ﾃﾝﾎﾟﾗﾘ実数	$-10^{18}-1$ から$10^{18}-1$(ﾊﾟｯｸ10進数) $-1.1x10^{4932}$ から $-3.4x10^{-4932}$ 0 $3.4x10^{-4932}$ から $1.1x10^{4932}$ （以上5行）実数

付録H　式と演算子

式は、定数値またはリロケータブルな値（オペランド）、またはその両方と、単一定数かリロケータブルな値を求める式演算子との組合せです。386｜ASM が認識する式演算子を表H－1に示します。

識別子への前方参照は、次の条件が成立した場合に式中で行うことができます。アセンブラがパス1で行う仮定により、アセンブラが必要とするバイト数のオブジェクトコードをパス1で生成するという条件です。正しくない前方参照の一般的理由は、セグメントオーバーライドバイトの生成を必要とする前方参照かFAR 型のラベルに対する前方参照です。正しくない前方参照は、通常PTR とセグメントオーバーライド（：）演算子を上手に使うことによって修正できます。ソースコードファイル中に前方参照が少ないほど、386｜ASM は速くそのファイルをアセンブルします。

式の評価順序は、演算子の優先順位により異なります。優先順位の最も高い演算子が最初に評価されます。優先順位の等しい演算子は左から先に評価されます。386｜ASM の式評価における演算子の優先順位を表H－2に示します。表中の同じ行にある演算子の優先順位は等しくなります。

表H－1　式演算子の概略

演算子	許容されるリロケータブルな値	形　　式	説　　明
( )	有	(expr)	括弧内の式計算を隣り合った演算より先に行う。
[ ]	有	expr [basereg] expr [indexreg] expr1　[expr2]	ベース付きまたはインデクス付きまたはベース付インデクス付アドレスモードを生成するのに使う。2つの式値を加えるためインデクス演算子としても使う。

演算子	許容される リロケータブルな値	形　　式	説　　　明
	有	expr.strfldname expr1.expr2	構造体フィールド名演算子は構造体内のフィールドオフセットを式に加え、その式のデータタイプをフィールドのデータタイプにセットする。 これはまた2式を加えるための＋演算子の代りに使われる。
:	有	segreg:expr segname:expr groupname:expr	セグメントオーバーライド演算子はexprのアドレスを指定のセグメントレジスタ、セグメントまたはグループ相対で計算するよう指定する。 またこの演算子は386｜ASM に必要があれば命令前にセグメントオーバーライドバイトを生成させる。名前のあるセグメントまたは名前のあるグループが、セグメントオーバーライド演算子で使われるより前にASSUMEディレクティブで、セグメントレジスタに割当てられなければならない。
／	無	expr1/expr2	小数部を切り捨てて整数除算する。
＊	無	expr1*expr2	整数乗算する。
＋	1オペランドのみ	expr1+expr2	加算する。
－	有	expr1-expr2	減算－両オペランドが同一セグメント相対である場合だけ、その両オペランドをリロケートできる。この場合結果は絶対数である。一方のオペランドがリロケータブルな場合はそのオペランドは左側になければならない。

演算子	許容される再配置可能値	形　　式	説　　　明
+(単項)	有	+expr	単項加算
-(単項)	無	-expr	単項減算
.TYPE	有	.TYPE expr	exprのモードとスコープを定義するバイトを返す。
AND	無	expr1 AND expr2	２つのオペランドのビットAND(論理積)
EQ	無	expr1 EQ expr2	両方の式が等しければ－１を返し、等しくなければ０を返す。
GE	無	expr1 GE expr2	expr1 がexpr2 以上であれば－１を返し、それ以外は０を返す。
GT	無	expr1 GT expr2	expr1 がexpr2 より大きければ－１を返し、それ以外は０を返す。
HIGH	無	HIGH expr	16ビット値のうちの上位８ビットを返す。
HIGHW	無	HIGHW expr	32ビット値のうちの上位16ビットを返す。
LE	無	expr1 LE expr2	expr1 がexpr2 以下であれば－１を返す。それ以外は０を返す。
LENGTH	有	LENGTHvarname	変数内のエレメント数を返す。ここで変数は、変数宣言文の中のDUP 演算子、またはイニシャライザリストの使用により複数のエレメントを持つ。
LOW	無	LOW expr	16ビット値のうちの下位８ビットを返す。

演算子	許容される再配置可能値	形　　式	説　　　明
LOWW	無	LOWW expr	32ビット値のうちの下位16ビットを返す。
LT	無	expr1 LT expr2	expr1 がexpr2 より小さければ－1を返し、それ以外は0を返す。
MASK	無	MASK recnm MASK recfldnm	レコード中のすべての定義されたビット、または特定フィールドが占めるレコード中のビットに対してビットマスクを返す。
MOD	無	expr MOD expr2	モジュロ演算子－整数除算後の余りを返す。
NE	無	expr1 NE expr2	両方の式が等しくなければ－1を返し、等しければ0を返す。
NOT	無	NOT expr	オペランドの1の補数
OFFSET	有	OFFSET expr	オペランドとそのオペランドを定義しているセグメントの始まりとの間のバイト数を返す。返された値は、再配置可能である。
OR	有	expr1 OR expr2	2つのオペランドのビットOR（論理和）
PTR	有	type PTR expr	式に指定データタイプを強制的に持たせる。ここでのタイプは、{BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR, FAR} のセットのうちの1つ、またはデータタイプに対応する整数値（TYPE演算子参照）である。

演算子	許容される再配置可能値	形　　式	説　　　　明
SEG	有	SEG expr	exprが関連するセグメントのセグメントセレクタを返す。結果はリロケータブルである。
SHL	無	expr1 SHL expr2	expr1 をexpr2 の値が指定するビット数だけ左にシフトする。式の終りからはみでたビットは失われる。
SHORT	有	SHORT label	アセンブラがJMP 命令中に１バイトのオフセットを生成するよう強制する。前方参照だけに必要。
SHR	無	expr1 SHR expr2	expr1 をexpr2 の値が指定するビット数だけ右にシフトする。式の終りからはみでたビットは失われる。
SIZE	有	SIZE varname	指定変数に割当られたバイト数を返す。(LENGTHvarname)*(TYPEvarname)と等しい。
THIS	無	THIS type	オフセット値とセグメント値がカレントロケーションカウンタ値に等しく、そのデータタイプが {TYPE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR, FAR} のセットのうち１つで示されるようなオペランドを作成する。
TYPE	有	THIS expr	exprのデータタイプを示す数字を返す。変数データタイプなら、そのデータタイプのサイズ（単位：バイト）が返される。ラベルデータタイプなら、NEARには－１を、FAR には－２が返される。

演算子	許容される 再配置可能値	形　　式	説　　　明
WIDTH	無	WIDTH recordname WIDTH recfldname	指定のレコードまたはレコードフィルドの幅（単位：ビット）を返す。
XOR	無	expr1 XOR expr2	２つのオペランドのビット排他的OR（論理和）

表Ｈ－２　演算子の優先順位

演　　算　　子	優　先　順　位
LENGTH, SIZE, WIDTH, MASK, [ ]	最　上　位
.（構造体フィールド名）	
:（セグメントオーバーライド）	
OFFSET, SEG, TYPE, THIS	
HIGH, LOW, HIGHW, LOWW	
単項 +, 単項 -	
PTR	
*, /, MOD, SHL, SHR	
+, -	
EQ, NE, LT, LE, GT, GE	
NOT	
AND	
OR, XOR	
SHORT, .TYPE	最　下　位

付録 I シンボルタイプ

ユーザ定義シンボルは、そのシンボルの定義方法によって暗黙に示されるデータタイプを持ちます。386 | ASM がサポートするシンボルタイプを表 I − 1 に示します。

表 I − 1　シンボルタイプ

シンボルタイプ	説　　明	シンボル作成用ディレクティブ
group	単数または複数のセグメントをグループにするのに使うグループ名	GROUP
segment	セグメント名。アライン型、組合せ型、プロテクションなどの属性を持つ。	SEGMENT
variable	特定セグメント中のデータのアドレス。BYTE, WORD, DWORD, PWORD, QWORD, TWORD のデータタイプを持つ。	DB, DW, DD, DQ, DP, DF, DT, LABEL, EQU, =
structure	構造体定義テンプレートを呼び出すことにより割当てられるデータ領域のアドレス。	構造体定義名
record	レコード定義テンプレートを呼び出すことにより割当てられるデータ領域のアドレス。	レコード定義名

シンボルタイプ	説　　　明	シンボル作成用ディレクティブ
label	JMP 命令またはCALL命令による制御トランスファに使うセグメント内のアドレス。NEARかFAR のデータタイプを持つ。	LABEL, =, EQU
procedure label	label とまったく同じ。	PROC
constsnt	整定数	EQU, =,
alias	アセンブラ予約語の別名	EQU
text substitution	シンボル名がどこで使われていても、そのシンボル名に代わって置き換えられるASCII のテキスト文字列	EQU
macro	マクロ定義名。呼び出されるとマクロを展開する。	MACRO
structure definition	構造体定義テンプレート名。呼び出されると、データ領域の割当てと初期化を行う。その領域のサイズと初期値はテンプレートの中に定義されている。	STRUC
structure field	構造体テンプレート中のフィールド名。データタイプ(BYTE, WORD, DWORD, PWORD, QWORD, TWORD)、構造体テンプレート内の、そのフィールドのオフセットおよび初期値を持つ。	構造体定義ブロック内の DB, DW, DD DQ, DP, DF DT
record definition	レコード定義テンプレート名。呼び出されると８ビット、16ビット、32ビットのデータ領域の割当てと初期化をする。	RECORD

シンボルタイプ	説　　　明	シンボル作成用 ディレクティブ
record field	レコードテンプレート中のビットフィールド名。レコードテンプレート内の、そのフィールドのオフセット（単位：ビット）、幅（単位：ビット）、および初期値を持つ	RECORD

付録J USE16セグメントとUSE32セグメントの混合

USE16セグメント内の命令は、デフォルトでは16ビットオペランドと16ビットアドレスモードを使います。USE 32セグメント内の命令は、デフォルトでは32ビットオペランドと32ビットアドレスモードを使います。しかし、オペランドサイズまたはアドレスモードのそれぞれの代替形を、独立には選ぶことができます。これにはオペランドサイズオーバーライドバイトかアドレスサイズオーバーライドバイトかのいずれか（またはその両方）を使って、その命令をプレフィックスします。386 | ASM は必要に応じてこれらオーバーライドバイトを自動的に生成します。この付録では16ビットと32ビット形の命令の場合に発生する問題と、USE 16とUSE 32のセグメントのプログラム間の対話問題についていくつか説明します。

J.1 ― セグメント間のプロシジャコール

使用型の異なるセグメント間でFAR プロシジャコールを行った時、386 | ASM は、デスティネーションセグメントの使用型に一致するCALL命令の形を生成します。これは、異なる使用型を持つセグメントからそのプロシジャを呼出したかどうかに関係なく、RET 命令のデフォルト形をいつも使用できることを意味します。

例：

```
cseg1   SEGMENT   use16
p1      PROC      far
        ret                 ; 16-bit far return

p1      ENDP
cseg1   ENDS

cseg2   SEGMENT   use32
        call      p1        ; assembler automatically
                            ; generates 16-bit form
                            ; of far call instruction
```

```
cseg2     ENDS

cseg3     SEGMENT   use16
          call      p1          ; 16-bit form of call

cseg3     ENDS
```

これは、USE 16セグメントからUSE 32セグメントへのFAR コール時にうまく動作します。CALLはフル32ビットのEIP をスタック上にプッシュし、RET (USE32　セグメント中の) がこのEIP を復元します。しかし、USE32 セグメントからUSE16 セグメントへのFAR コールには1つの問題がひそんでいます。16ビット形のCALLとRETを使うので、IP (EIPでなく) だけがスタック上に保存されて、復元されます。このように何かのコードがCALLとRET 間で、EIP の上位16ビットを変更する場合、RET は制御を正しい位置に戻しません。通常、このことは問題になりません。その理由は、USE16 セグメント内での実行時、EIP の上位16ビットを変更する理由はないからです。もしこの上位16ビットの変更があり得るなら対策はあります。386 | ASM は間接コール（本付録第2節参照）を使って32ビット形のCALLを強制的に生成することができます。32ビット形のRET は、RET 命令の前にオペランドサイズオーバーライドバイトをハードコードすることによって強制的に生成できます。これを実行した場合、プロシジャはもう16ビット形のCALL命令ではコールできないことに注意してください。

例：

```
cseg1         SEGMENT   use16
p1            PROC      far
              DB        66h         ; operand size override
                                    ; gives 32-bit form
                                    ; of return
p1            ENDP
cseg1         ENDS

dseg          SEGMENT
p1_call       LABEL     pword       ; indirect 32-bit far call
              DD        OFFSET p1
              DW        cseg1

dseg          ENDS

cseg2         SEGMENT   USE32
              call      p1_call     ; force 32-bit form
                                    ; of call
```

```
cseg2           ENDS

cseg3           SEGMENT   USE16
                call      pl                ; this won't work
                                            ; because of 32-bit
                                            ; return in proc pl
cseg3           ENDS
```

J.2 — 間接制御トランスファ

386 ネィティブドモードの間接コールおよび間接ジャンプは、同一プログラム中にUSE16 とUSE32 のセグメントを混合する可能性があるため、アセンブラに対して曖昧さの問題を提示します。USE16 セグメントから間接コール／間接ジャンプを行う時、この間接コール／ジャンプと一緒に使用中の変数のデータタイプを、どの形の命令を生成するかを決めるのに使用できます。

変数データタイプ	コール／ジャンプ
WORD	USE16 ｾｸﾞﾒﾝﾄへのNEARコール/NEARジャンプ
DWORD	USE16 ｾｸﾞﾒﾝﾄへの FARコール/FARジャンプ
PWORD/FWORD	USE32 ｾｸﾞﾒﾝﾄへの FARコール/FARジャンプ

しかし、USE32 セグメントからの間接コール／間接ジャンプをする時は、データタイプは曖昧です。

変数データタイプ	コール／ジャンプ
WORD	USE16 ｾｸﾞﾒﾝﾄへの FARコール/FARジャンプ
DWORD	USE32 ｾｸﾞﾒﾝﾄへのNEARコール/NEARジャンプ
PWORD/FWORD	USE32 ｾｸﾞﾒﾝﾄへのFARコール/FARジャンプ

したがって、386 | ASM は変数がWORD型でUSE32 セグメントで実行中の、間接コール／間接ジャンプはUSE16 セグメントへのFAR コール/FARジャンプを意味するという規約を使います。データタイプが間接呼出し／間接ジャンプに対して生成された命令のタイプを決める方法のあらましを表J－1に示します。USE32 セグメントからの３種類すべての間接制御トランスファを作成する例を次に示します。

例：

```
data      SEGMENT
far32     DP        ?                   ; seg:offset in USE32 seg.
near32    DD        ?                   ; offset in USE32 seg.
far16     DD        ?                   ; seg:offset in USE16 seg.
data      ENDS

code      SEGMENT   use32
          call      far32               ; FAR call to USE32 seg.
          jmp       near32              ; NEAR jump within seg.
          call      word PTR far16      ; force FAR call to USE16 seg.
code      ENDS
```

表J－1　間接制御トランスファデータタイプ

コールするセグメントの使用型	変数データタイプ	生成される制御トランスファの型
USE16	WORD DWORD PWORD/FWORD 16ビットレジスタ 32ビットレジスタ	USE16ｾｸﾞﾒﾝﾄ内のNEARコール/NEAR ジャンプ 別のUSE16ｾｸﾞﾒﾝﾄへのFAR コール/FARジャンプ USE32ｾｸﾞﾒﾝﾄへのFAR コール/FARジャンプ USE16ｾｸﾞﾒﾝﾄ内のNEARコール/FARジャンプ 誤り
USE32	WORD DWORD PWORD/FWORD 16ビットレジスタ 32ビットレジスタ	USE16ｾｸﾞﾒﾝﾄへのFAR コール/FARジャンプ USE32ｾｸﾞﾒﾝﾄ内のNEARコール/NEAR ジャンプ 別のUSE32ｾｸﾞﾒﾝﾄへのFAR コール/FAR ジャンプ 誤り USE32ｾｸﾞﾒﾝﾄ内のNEARコール/NEAR ジャンプ

J.3 － スタックPUSH/POP用のデータ幅

80386 プロセッサは、WORD値(16ビット) かDWORD値(32ビット) のうちのいずれかをスタック上にプッシュします。PUSH/POP用のデフォルトオペランドサイズは（すべての命令について）、USE16 セグメントで実行中の時WORD, USE32セグメントの場合はDWORD です。PUSH/POP命令へのオペランドがデータタイプを持たない場合、386 ｜ASM はオペランドサイズオーバーライドバイトを生成しません。そのため、デフォルトオペランドサイズはプッシュかポップされます。PTR 演算子は、アセンブラに望ましいオペランドサイズを使わせるために、DWORD またはWORDK データタイプをオペランドに与えるのに使用できます。

例：

```
cseg    SEGMENT  use32
        push     0              ; push a DWORD
        push     word PTR 0     ; push a WORD
        push     dword PTR 0    ; push a DWORD
cseg    ENDS
```

セグメントレジスタのプッシュとポップを行う場合、80386 は、USE16 セグメントで実行している時は、16ビットを、そしてUSE32 セグメントでの実行時は32ビット（このうち上位16ビットは使用しないで0 がプッシュされる）をプッシュ/ ポップします。デフォルトオペランドサイズの変更には、オペランドサイズオーバーライドバイトを使うことは許されます。しかし、386｜ASM はオーバライドバイトを自動的に作成する方法を持っていません。オーバーライドバイトが必要な場合、そのオーバーライドバイトはDBを使ってコード化します。

例：

```
cseg16  SEGMENT  use16
        push     ds             ; push 16 bits
        pop      ax
cseg    ENDS
cseg32  SEGMENT  use32
        push     eax
        pop      ds             ; pop 32 bits
        DB       66h            ; operand size override
        push     ds             ; push 16 bits
cseg32  ENDS
```

PUSHF/POPF（プッシュ／ポップフラグ）の命令にも２つの形があります。16ビット形では、EFLAGSレジスタの下位16ビットをプッシュ／ポップします。32ビット形ではEFLAGSレジスタの全32ビットをプッシュ／ポップします。希望する命令形の生成にはどのような命令ニーモニックを使用したらよいかを表Ｊ－２に示します。プログラマは、命令ニーモニックの選択によって、どの形の命令を生成するかをはっきり指定できます。

表Ｊ－２　　プッシュ命令／ポップフラグ命令

ニーモニック	生成される命令
PUSHF	EFLAGSの下位16ビットをプッシュする。
POPF	WORDをEFLAGSの下位16ビットにポップする。
PUSHFD	EFLAGSの全32ビットをプッシュする。
POPFD	DWORD をEFLAGSにポップする。

```
例：    cseg16    SEGMENT  use16
                  pushf    ; push low 16 bits of EFLAGS
                  popfd    ; pop DWORD into EFLAGS
        cseg16    ENDS
        cseg32    SEGMENT  use32
                  pushfd   ; push all 32 bits of EFLAGS
                  popf     ; pop WORD into low 16 bits of EFLAGS
        cseg32    ENDS
```

J.4 ― 割込み復帰命令

割込み復帰（IRET）命令は、32ビットと16ビットの形を持ちます。386 ネイティブモード割込みハンドラは、必ず32ビット形を使い、リアルモード割込みハンドラは必ず16ビット形を使います。386|ASM は、IRETD ニーモニックの使用時、32ビット形の命令を生成し、IRETニーモニックの使用時には、16ビット形の命令を生成します。

```
例：   cseg      SEGMENT        use16

                 iret                           ; interrupt return for
                                                ; real mode handler

                 iretd                          ; interrupt return for
                                                ; protected mode handler

       cseg      ENDS
```

J.5 ― ロード／ストアディスクリプタテーブルレジスタ

GDTRレジスタとIDTRレジスタのためのロード命令およびストア命令には、16ビット形と32ビット形とがあります。16ビット形は、80286 との互換を保つため設けたものであり、ディスクリプタテーブルベースアドレスの24ビットのロードとストアだけを行います。32ビット形は、全32ビットベースアドレスのロードとストアを行います。386|ASM はサフィクス“E”を命令ニーモニックと共に使う時、必ず32ビット形の命令を生成し、“E”を使わない時は、24ビット形の命令を生成します。どんな命令ニーモニックが、どんな形の命令を生成するかを表Ｊ－３に示します。

表J－3　ディスクリプタテーブルレジスタ命令

ニーモニック	生成される命令
LGDT/SGDT	16ビット(80286) 形のロード／ストアGDTR
LGDTE/SGDTE	32ビット(80386) 形のロード／ストアGDTR
LIDT/SIDT	16ビット(80286) 形のロード／ストアIDTR
LIDTE/SIDTE	32ビット(80386) 形のロード／ストアIDTR

付録K 簡易OMF－386 オブジェクトファイルフォーマット

簡易OMF-386 は、 386|ASM が80386 用に生成されたオブジェクトファイルに使用するフォーマットです（8086、8088、80186、80286 用に生成されるオブジェクトファイルはOMF-86フォーマットです）。簡易OMF-386 は、Intel およびMicrosoft によって使用されるOMF-86フォーマットを単純に拡張したものです。この付録では、簡易OMF-386 とOMF-86との相違を説明します。OMF-386 についてはリファレンス１と２を参照してください。

386|ASM は、オブジェクトモジュールの始めに次のコメントレコードを置くことによって、オブジェクトモジュールが80386 のターゲットであることをリンカに知らせます。

0	1	2	3 … 7	8
88H	80H	AAH	'80386'	
レコード型	フラグ	クラス		チェックサム

'H' は１６進値を意味する。

80386 コメントレコードは、モジュール見出しレコード（THEADR）の直後で、そしてオブジェクトモジュールの他のレコードより前に置きます。

オブジェクトモジュールの他のレコードは、OMF-86と同じような方法で、簡易OMF-386 に、フォーマットされます。ただし、異なるのは、簡易OMF-386 ではオブジェクトレコードのオフセット、ディスプレースメントまたはセグメント長フィールドは２バイトではなく４バイトである点です。次のレコードは、サイズを大きくしたフィールドを持ちます。

レコード	フィールド
SEGDEF	オフセット長およびセグメント長
PUBDEF	オフセット
LEDATA	オフセット
LIDATA	オフセット
Explicit FIXUPP	ターゲットディスプレースメント
BLKDEF	復帰アドレスオフセット
LINNUM	オフセット
MODEND	ターゲットディスプレースメント
DEBSYM	オフセット
LOCSYM	オフセット

FIXUPPレコードにおいては、次の２つ新しい"Loc" 値が定義されています。

5	32ビットオフセット
6	ベース＋32ビットオフセット（ロングポインタ）

SEGDEFレコードにおいては、次の２つ新しい"Align" 値が定義されています。

5	リロケータブルダブルワードアラインセグメント（セグメントはダブルワード境界で開始される）
6	リロケータブル4Kページアラインセグメント（セグメントは4096バイト境界のアドレスで開始される）

付録 L　386|ASM コマンドライン例

1．オブジェクトファイル"HELLO.OBJ"とリスティングファイル"HELLO.LST"を作成するためにファイル"HELLO.ASM"をアセンブルする。

```
386ASM HELLO
```

2．ファイル"HELLO.ASM"をアセンブルし、リスティングファイルを生成しない。

```
386ASM HELLO -NOLIST
```

3．ファイル"HELLO.ASM"をアセンブルし、オブジェクトファイルを生成しない。

```
386ASM HELLO -NOOBJECT
```

4．8086用にファイル"HELLO.ASM"をアセンブルする。

```
386ASM HELLO -8086
```

5．ファイル"HELLO.ASM"をアセンブルし、リスティングとオブジェクトファイルを別のディレクトリに転送する。

```
386ASM HELLO -LIST ¥LISTINGS¥HELLO -OBJECT ¥OBJS¥HELLO
```

6．ファイル"HELLO.ASM"をアセンブルし、すべてのエラーをファイル"HELLO.ERR"に書き込む。

```
386ASM HELLO -ERRORLIST HELLO
```

索引

386 | ASM TOOL KIT ユーザーズマニュアル
第１部 アセンブラ 386 | ASM

英数・記号

あ行

か行

さ行

た行

は行

ま行

や行

ら行

わ行

第2部

リンカ

386 | LINK

目次

386|ASM TOOL KIT ユーザーズマニュアル
第2部 リンカ 386|LINK

第1章 概要

386 | LINKはインテル80386 マイクロプロセッサ用のリンカです。 386 | LINKは、アセンブラまたはコンパイラで作成された別々のオブジェクトモジュールを、単一の 80386アプリケーションに組み合わせるのに使用されます。 386 | LINKは、80386 用のアセンブラである 386 | ASM、あるいは他のメーカの 80386高級言語コンパイラによって生成されたオブジェクトモジュールをリンクすることができます。

386 | LINKは、8086および80286 のマイクロプロセッサ用のプログラムをリンクすることもできます。本書では 386 | LINKの使用方法について説明します。

第2章 386 | LINKの使用

2.1 プログラムのリンク

386 | LINKを使ってプログラムをリンクするには、コマンド "386LINK"のあとに、(たとえば以下のように) プログラムを構成しているオブジェクトファイルを指定します。

```
386LINK MAIN SUBA SUBB SUBC
```

スペースまたはカンマにより、オブジェクト入力ファイル名を区切ります。出力プログラムファイル名は、特に指定のない限り、ファイル名拡張子を ".EXE" とし、最初のオブジェクト入力と同じ名前を与えます。たとえば、オブジェクトファイル "HELLO.OBJ" をリンクすると、"HELLO.EXE" というプログラムファイルが作成されます。

さらにリンカは、マップファイルを出力し、これをプリント、または画面に表示することができます。マップファイルは、プログラムを構成するためのリンク対象ファイル、プログラム内のセグメント位置とサイズ、パブリックシンボルの位置を示しています。マップファイルのフォーマットは本章の 2.5節で詳しく説明します。

オブジェクトファイルにファイル名拡張子がない場合は 386 | LINKが".OBJ"をとります。386 | LINKは、オブジェクト入力ファイルを順次読み取り、異なるオブジェクトファイル間の外部シンボルの参照を解決し、一つのプログラムファイルとして書き出します。リンカは、デフォルトではMS-DOSの.EXEファイルを出力ファイルとして書きます。オプションとして、リンカは、プログラムをIntel hex ファイルとして出力することができます。

2.2 コマンドスイッチ

リンカ操作の制御は、コマンドスイッチを使って行います。コマンドスイッチはコマンドラインで指示します。スイッチはすべてマイナス記号 "-"で始まります。ファイル名はマイナス記号で始めることができません。したがってリンカは、ファイル名かスイッチかをマイナス記号の有無で区別します。スイッチとファイル名はスペースで区切ります。本節では 386 | LINKの基本的なスイッチについて説明します。

通常、リンカはマップファイルを作成して、プログラムのデバッグを助けます。プログラムがデバッグ済みの場合、マップファイルを必要としないことがあります。マップファイルを作成をしないときは、次のようにスイッチの"-NOMAP"を指定します。

```
386LINK HELLO -NOMAP
```

"-NOMAP"スイッチは"-NOM"と省略できます。

リンカがマップファイルを作成するときには、マップファイル名として最初のオブジェクト入力ファイル名を使用し、".MAP"の拡張子を付けます。この名前が適切でない場合には、スイッチの"-MAP"を使ってマップファイル名を指定します。マップファイル名は、次のようにスイッチのあとに指定します。

```
386LINK HELLO -MAP JULY31
```

"-MAP"コマンドスイッチの他の使用法は、マップファイルを別のディスクに転送することです。マップファイルの容量は非常に大きいので、オブジェクト入力ファイル、.EXEファイルとマップファイルを同じディスクに収容できないことがあります。次のように、"-MAP"スイッチで別のディスクにマップファイルを書き込むことができます。

```
386LINK HELLO -MAP A:HELLO
```

"-MAP"スイッチは、"-M"と省略できます。

プログラムの最初の立上げ時に、リンクのエラーにより、プログラムを実行できず、何度か再リンクの操作が必要なことがあります。このような状況において、未定義の外部シンボルなどのエラーを探しながら試しにリンクするときは、.EXEファイルを生成する必要をあまり感じません。これらのトライアルリンクでは、"-NOOUTPUT"スイッチを使って、386|LINKに対して.EXEを生成しないよう指示できます。

"-NOOUPUT"は"-NOO"と略すことができます。

マップファイルと同様に、リンカが生成する.EXEファイルをリネーム（名前変更）したり他のディスクに移すことができます。"-EXE"スイッチは、リンカが生成した.EXEファイルの名前または位置を明示的に記述するのに使用されます。このスイッチの後ろには、.EXEファイルに与えられる名前を続けなければなりません。

例：

```
386LINK HELLO -EXE NEWHELLO
386LINK HELLO -EXE A:HELLO
386LINK HELLO -EXE ¥PROGRAMS¥HELLO
386LINK HDR MAIN SUBA SUBB SUBC -EXE MAIN
```

2.3　ライブラリファイル

高級言語を使って80386 マイクロプロセッサ上でプログラムする場合、コンパイラのメーカは実行時ライブラリもコンパイラに含めて提供します。実行時ライブラリには、どのようなプログラムからでも呼び出せ、共通に使える関数が含まれています。実行時ライブラリ中の一般的な関数を以下に示します。

－プログラム初期化コード
－ファイル入出力関数
－浮動小数点演算関数
－文字列操作関数
－フォーマット関数

これらの関数は全部で数百個ありますが、ライブラリファイルと呼ばれる単一のファイルにまとめられます。次にそのライブラリファイルは、コンパイラによってコンパイルされたプログラムとリンクされます。

一つのプログラムが、実行時ライブラリのすべての関数を使用することはほとんどありません。したがって、リンカがライブラリファイルのすべての関数を一つのプログラムにリンクするというのは良い考えとはいえません。その理由は、不要な関数がプログラムのサイズを極端に大きくするからです。したがって、ライブラリファイルとのリンク時、386|LINKは、ユーザプログラムや既にリンクされている他のライブラリ関数から直接呼び出されるライブラリ関数をユーザプログラムにインクルードするだけです。

ライブラリファイルをリンクしていることをリンカが分かるように、そのライブラリファイルの名前が、すべてのオブジェクト入力ファイルの後に "-LIB" スイッチで与えられます。

たとえば、"EDIT"という名前のC言語プログラムを持っているとします。このプログラムをC言語実行時ライブラリとリンクするには、次のリンクコマンドを使います。

```
386LINK EDIT -LIB CLIB
```

"CLIB" はユーザの実行時ライブラリの名前であると仮定します。コンパイラと一緒に提供されるドキュメントには、ユーザがリンクしなければならない実行時ライブラリ名が記されています。

また、複数のライブラリファイルとリンクすることも可能です。その場合、"-LIB"スイッチのあとに各ライブラリファイルの名前をリストします。

```
386LINK PAINT -LIB CLIB GRAPHICS MATH SCREEN MOUSE
```

ライブラリファイル名を区切るにはスペースかコンマを使用します。

MS-DOSシステムでは、ライブラリファイルのデフォルトファイル名拡張子は ".LIB" です。

2.4 リンカコマンドファイル

リンカコマンドラインが長すぎる場合、コマンドファイルからの読込みをリンカに指示できます。スイッチとオブジェクトファイルの名前を、コマンドラインに入力する代わりに、ファイルに入力します。ファイル名とスイッチは、コマンドファイル内では必要なだけ行数を使って書くことができます。たとえば次のコマンドライン

```
386LINK MAIN SUB1 SUB2 SUB3 -LIB CLIB
```

は、以下のコマンドファイルで表すことができます。

```
MAIN
SUB1 SUB2 SUB3
-LIB CLIB
```

コマンド名 "386LINK" はコマンドファイルには指定できないことに注意してください。
コマンドファイルを使ってリンカを実行するには次のように入力します。

```
386LINK @xxx
```

"xxx" はコマンドファイルの名前です。この例でコマンドファイルの名前が"MYLINK"の場合、

```
386LINK @MYLINK
```

とするとそのファイルをリンクします（注：リンカはリンカコマンドファイルのファイル名拡張子として ".LNK" を想定しており、実際のファイル名は "MYLINK.LNK" となる）。

コマンドファイルからリンクする場合、コマンドラインにはコマンドファイル名以外の他のファイル名またはコマンドスイッチを入力することは許されません。
コマンドファイルは、単に入力を減らすだけの目的で使用することもできます。リンカコマンドラインは、一つのファイルに最低一度は入力しなければなりませんが、入力すれば何度でも使用できます。

2.5 マップファイル

リンカが生成するマップファイルは道路地図のようなものです。基本的には、デバッグ中に、セグメント、関数、および変数の位置を知るために使用されます。特に指定のない場合、リンカは必ずマップファイルを生成します。デフォルトでは、マップファイルの名前は、拡張子".MAP"を持つ最初のオブジェクトファイルの名前です。

次に、リンカが生成する一般的なマップファイルを示します。

```
386 | LINK:1.1   -- Copyright (C) 1986 Phar Lap Software Inc. ……(1)

Target CPU is 80386 ………………………………………………………………(2)

Input module(s):……………………………………………………………………(3)

"MAIN" from file "MAIN.OBJ".
"TRIG" from file "TRIGSUBS.OBJ".

Segment map ……………………………………………………………………………(4)

Name      Group  Class   Type  Offset    Size
-------   -----  ------  ----  --------  --------
_TEXT     CODE           PUB   00000000  000003DA
_DATA     DATA           PUB   000003E0  00000054
_STACK    STACK          STK   0000042E  00002000

Public symbols………………………………………………………………………………(5)

Name      Value     Module   Segment   Size
-------   --------  -------  ------    -------
ATAN      0000019A  TRIG     _TEXT
COS       00000080  TRIG     _TEXT
MAIN      00000000  MAIN     _TEXT
PI        00000426  TRIG     _DATA
SIN       00000085  TRIG     _TEXT
TAN       0000016F  TRIG     _TEXT
```

マップの主要項目を以下に示します。

(1) このメッセージ行は、実行中の 386 | LINKのバージョンを示します。このバージョン番号は、何か問題があってPhar Lap Software 社に連絡するときに重要となる番号です。

(2) 2行目は、このプログラムがターゲットとするインテル"86"CPU ファミリのメンバを示しています。このプログラムは 386ネイティブモードプログラムであるため、ターゲットCPU は 80386です。後述するように、 386 | LINKは、 8086CPUおよび80286CPU用プログラムもリンクできます。

(3) ここはプログラムを構成するためにリンクされるモジュールを表示しています。モジュール名およびモジュール読込み元のファイルが表示されます。モジュール名は、 386 | ASM のNAMEディレクティブかTITLE ディレクティブのいずれかで設定されます。

(4) ここは、プログラムを構成する各種のセグメントを表示しています。セグメントごとに次の情報が示されています。
　－セグメント名
　－セグメントが属するグループ名（ない場合もある）

－SEGMENT ディレクティブで指定されたときのセグメントのクラス
－SEGMENT ディレクティブで指定されたときのセグメントのタイプ
有効なセグメントタイプを以下に示します。

ABS : Absolute　　PUB : Public　　STK : Stack
COM : Common　　PVT : Private

－プログラム開始点からのセグメントの開始オフセット
－セグメントのサイズ（単位：バイト）

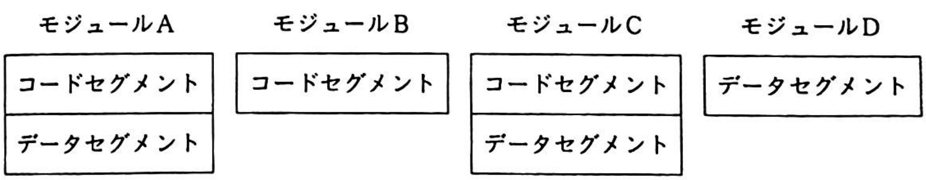
ほとんどの場合、デバッグ中はセグメントマップは無視できます。セグメントマップの一つの使い方としては、セグメントに分割されたプログラムのサイズを見ることがあります。セグメントマップにより、プログラムのコード部分のサイズをデータのサイズと比較することができます。

(5) ここにはユーザプログラムで定義されているすべてのパブリックシンボルの名前と位置が表示されています。シンボルはアルファベット順に表示されています。パブリックシンボル定義のメモリ内の位置はシンボルの後に表示されています。また、シンボルが定義されているモジュール名、およびセグメント名も表示されています。

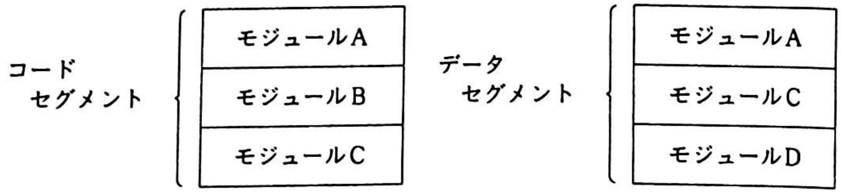
パブリックシンボルリストは、デバッグにとって最も重要な部分です。シンボルに関してリストされたアドレスは、デバッグしてブレイクポイントをラベルにセットするか、変数の内容を調べるときに使用できます。
パブリックシンボルの後ろには、未定義だがプログラムで参照された外部シンボルがリストされます。このプログラム例には未定義の外部シンボルがないため、何もリストされません。
リンク時にエラーが発生した場合、それらは画面に表示されます。さらに、エラーメッセージがマップファイルにも書き込まれます。エラーメッセージは常にオブジェクト入力ファイルのリストの直前に置かれます。

2.6 セグメンテーション

386｜LINKの仕事の一つに、別々のオブジェクトモジュールから発生した異なるセグメント部分を一つのセグメントに組み合わせることがあります。通常の80386 プログラムでは、80386 命令用のセグメント（コードセグメント）とデータ用の別のセグメント（データセグメント）があります。プログラムを構成している各モジュールにはコードセグメント部分とデータセグメント部分が含まれています。これらの部分は、単一のコードセグメントと単一のデータセグメントに組み合わせなければなりません。たとえば、プログラムがA，B，CおよびDの四つのモジュールから構成されている場合、モジュール内のセグメントのレイアウトは次のようになります。



しかし、リンカはこれらのセグメント部分を以下のように、単一のコードセグメントと単一のデータセグメントとに再構成しなければなりません。



386 | LINKは、セグメントの新しい部分をオブジェクトモジュールから読み込む場合、新しい部分を 386 | LINKが生成する出力プログラムファイルに挿入しなければなりません。386 | LINKは以下のルールに従って、新しい部分を出力ファイルに挿入します。

1. 同じ名前を持つセグメントが既に出力ファイル内に存在する場合、新しい部分はそのセグメントの最後に置かれます。
2. 1.以外の場合、同じセグメントクラスのセグメントがあれば、新しい部分は、同じクラス名を持つセグメントのうちの最後のセグメントの次に新しいセグメントを開始します。
3. 1.でも2.でもなくて、セグメントがヌル以外のクラス名を持っている場合、その新しい部分はファイルの終りで、ヌルクラス名を持つセグメントの前に新しいセグメントを開始します。
4. 1.でも2.でも3.でもなければ、新しい部分は、新しいセグメントをファイルの終りから開始します。

リンカに各セグメントを指定の順序に並べるように強制することができます。これは、セグメントを希望の順序でリストするセグメント定義だけを持つ、短いアセンブリ言語モジュールを記述することで行ないます。このモジュールを他のモジュールより前にリンクすれば、リンカは指定された順序にセグメントを並べます。たとえば、次のファイルは最初にコードセグメント、その次にデータセグメント、最後にスタックセグメントを挿入します。

```
_code    segment public byte use32  'code'
_code    ends

_data    segment public byte use32  'data'
_data    ends

_stack   segment public byte use32  'stack'
_stack   ends

         end
```

2.7 インテル16進出力

ユーザプログラムをMS-DOSでロードするのではなく、 ROM（読込み専用メモリ）から実行する場合、ユーザプログラムをインテル16進ファイルとしてリンカに出力させる必要があります。インテル16進ファイルは、ほとんどのPROMプログラマによって認められています。

386|LINKに.EXEファイルではなく16進ファイルを出力させるには、次に示すように、ファイル名の前にスイッチ"-HEX"を使用します。

```
386LINK BIOSROM -HEX BIOSROM
```

スイッチ"-HEX"は"-H"と略すことができます。
MS-DOSシステムでは、16進ファイルのデフォルトファイル名拡張子は".HEX"です。

2.8 8086、80286あるいはリアルモードのプログラムのリンク

386|LINKはまた、インテル"86"マイクロプロセッサファミリの他のメンバ用のプログラムをリンクします。プログラムを8086、8088、80188、あるいは80186のCPU上で実行する場合は、コマンドラインにスイッチ"-8086"を指定しなければなりません。80286または80386のリアルモードで走るプログラムをリンクする場合も、スイッチ"-8086"を使用しなければなりません。リアルモードは80286マイクロプロセッサと80386マイクロプロセッサの特長であり、既存の8086アプリケーションを新しいチップ上で実行可能にします。

ユーザプログラムを80286システムのプロテクトモードで実行する場合は、スイッチ"-80286"を指定します。

デフォルトでは、リンカは、ユーザプログラムを386ネイティブモードで実行することを想定します。リンカコマンドファイルを読んでいる他のユーザに対して、そのユーザプログラムが386ネイティブモードをターゲットにしていることを明らかにするために、スイッチ"-80386"を指定することもできます。

第3章 コマンドラインリファレンス

本章は、386|LINKのコマンドライン引数のリファレンスガイドであり、各引数がどのように処理されるかを詳細に説明します。引数は、オブジェクト入力ファイル、コマンドスイッチ、あるいはリンカコマンドファイル指定子になることができます。各引数に対しては以下の情報が与えられます。

- 引数の構文
- 引数の目的の説明
- 引数の使用例
- 引数がスイッチの場合の正しい省略形

3.1 入力ファイル

構文： file1 file2 file3...
または file1, file2, file3,...

説明： リンカコマンドに対する最初のオペランドは、リンクするオブジェクトファイル名のリストです。ファイル名はスペースまたはコンマで区切ることができます。ファイルはそのリストに与えられた順にリンクされます。ファイルにファイル名拡張子が指定されてない場合は、デフォルトの ".OBJ" になります。リンカのコマンドスイッチはすべてオブジェクトファイル名のあとに続けなければなりません。オブジェクト入力ファイルの指定が何もなければ、リンカは実行されません。

例：

```
386 LINK MAIN SUB1 SUB2 SUB3
386 LINK MYSTUFF, MYSUBS
386 LINK HELLO
```

3.2 オンラインヘルプ情報

構文： 386 LINK

説明：　リンカスイッチの名前あるいはフォーマットを忘れた場合、空のコマンドラインでリンカを実行させることにより、オンラインヘルプ情報を得ることができます。オンラインヘルプ情報は、有効なスイッチとそのフォーマットのリストを画面に表示します。

例：　386 LINK

```
             Phar Lap 386|LINK -- Version 1.1
             (c) 1986 Phar Lap Software Inc.
Command line options:
386LINK file1,file2,...          Input files  [.OBJ]
        -LIB file1,file2,...     Library files  [.LIB]
        -EXE file                .EXE output file [.EXE]
        -HEX file                Intel hex output file
        -NOOUTPUT                No output file
        -MAP file                Map file [.MAP]
        -MINDATA size            Set minimum data size
        -MAXDATA.size            Set maximum data size
        -STACK size              Set stack size
        -NOMAP                   No map file
        -DUMP                    Dump object files
        -ONECASE or -TWOCASE     Select case of symbols
        -8086 -80286 -80386      Select target CPU
```

3.3　リンカコマンドファイル

構文：　@file

説明：　リンカは通常、オブジェクト入力ファイル名とそのコマンドスイッチをコマンドラインから取り込みます。コマンドラインに収容すべき情報が多過ぎると、リンカは代わりに、オブジェクト入力ファイル名とコマンドスイッチをコマンドファイルから読み込むようにできます。コマンドファイル名を指定するには、リンカコマンドラインのコマンドファイル名の前に"@"を入力します。"@"とファイル名との間をスペースにすることは許されません。たとえば、

386LINK @MYLINK

コマンドファイルにファイル名拡張子が与えられてない場合、拡張子は".LNK"とみなされます。コマンドファイル名の指定がある場合、他のコマンドスイッチもオブジェクト入力ファイル名もコマンドラインに置くことは許されません。これらはすべてコマンドファイル内に置かなくてはなりません。コマンドファ

イルは、ホストシステムの標準エディタを使ったテキストファイルとして作成されます。オブジェクトファイルとコマンドスイッチは、コマンドラインに書かれるのとまったく同じようにコマンドファイルに書き込まれます。しかしながら、コマンドラインが1行だけであるのに対して、コマンドファイルは、ファイル名とスイッチを複数行に書き込むことが許されます。コマンドファイル中の必要なだけのファイル名およびスイッチがプログラムをリンクするために使えます。コマンドファイルの改行はちょうどスペースのように作用し、ファイル名またはコマンドスイッチを区切ります。コマンドファイルの入れ子は許されません。

例：　386LINK @MYLINK で、ファイル"MYLINK.LNK"に次のものが含まれています。

```
MAIN SUB1 SUB2 SUB3 SUB4
PROCL SCREEN KEYBOARD
DISK
-LIB SCRLIB CLIB KYBLIB IOLIB
-EXE SPREAD
-MAP SPREAD
-ONECASE
```

3.4 -DUMP コマンドスイッチ

構文：　-DUMP

説明：　スイッチ"-DUMP"は、リンカに対してマップファイル中にオブジェクト入力ファイルをプリントすることを伝えます。オブジェクト入力ファイルは16進でダンプされます。この機能は、386 | LINKと他メーカが提供するコンパイラとの間の互換性問題をデバッグするのに有用です。
注：-NOMAPの指定がある場合、オブジェクトファイルダンプをプリントするマップファイルがないため、"-DUMP" を使っても意味がありません。

例：　386LINK HELLO -DUMP（HELLO.OBJをダンプ出力する）

省略形：なし

3.5 -EXEコマンドスイッチ

構文：　-EXE

説明：　-EXEスイッチにより、リンカが生成される.EXEファイルの名前と位置を指定することができます。.EXEファイルにはユーザがリンクしたプログラムが記憶されています。通常、リンカは、最初のオブジェクト入力ファイルと同じ名前を.EXEファイルに与えます。ただし、ファイル名拡張子は".EXE に変わります。

.EXEファイルは最初のオブジェクト入力ファイルと同一のディスクとディレクトリに置かれます。このデフォルトの.EXEファイル名が受け入れられない場合には"-EXE"スイッチを使ってその名前を変えることができます。"-EXE"スイッチのあとには新しい.EXEファイル名を置かなければなりません。

例：　386 LINK HELLO -EXE ¥PROGLIB¥HELLO

省略形：-EXEの代わりに-E

3.6 -HEXコマンドスイッチ

構文：　-HEX file

説明：　リンカは通常、リンクされたプログラムをMS-DOSの.EXEファイルとして出力します。リンカはまた、プログラムをインテル絶対16進ファイルとして出力できます。
多くのPROMプログラマおよびCPU エミュレータがインテル絶対16進ファイルを入力ファイルとして受け付けます。リンカに16進ファイルを出力させるには、後にファイル名が続く-HEXスイッチを与えます。ファイル名拡張子が与えられてない場合は、".HEX"が拡張子とみなされます。

例：　386 LINK BIOSROM -HEX BIOSROM

省略形：-HEXの代わりに-H

3.7 -8086, -80286, -80386 のコマンドスイッチ

構文：　-8086
-80286
-80386

説明：　386 | LINKは、80386マイクロプロセッサ用のプログラムをリンクする以外に、8086および 80286のマイクロプロセッサ用のプログラムもリンクします。リンクされたプログラムを8086、80186、あるいは8088のシステム上で実行するつもりなら、リンク時に"-8086" コマンドスイッチを指定します。プログラムを80286 システムのプロテクトモードで実行する場合は、"-80286"コマンドスイッチを指定します。80286のリアルモードで実行する場合は、"-8086" を指定します。その理由は、80286上のリアルモードがちょうど8086のように動作するからです。"-80386"スイッチは、ターゲットCUP が 80386であることを指定します。リンカのデフォルト指定は 80386になっているので"-80386"はあえて指定する必要はありません。

例：　386 LINK MSCFIXS SIEVE -LIB　¥MSC¥SLIBC -8086
（MS-DOS用のマイクロソフトCプログラムをリンクします。）
386LINK NEWDOS -80286

省略形：-8086 の代わりに-86
-80286の代わりに-286
-80386の代わりに-386

3.8　-LIBコマンドスイッチ

構文：　-LIB file1 file2 file3...

説明：　コマンドスイッチ-LIBは、オブジェクトライブラリファイルのリストをリンカに指定するために使われます。ライブラリファイル名は"-LIB"スイッチの後に続けます。ファイル名同士を区切るにはスペースまたはコンマが使われます。ファイル名に拡張子がない場合は、リンカは拡張子を".LIB"とみなします。

386|LINKは、マイクロソフトのライブラリマネジャLIB によって作成されたライブラリファイルを受け付けます。ライブラリは、異なるプログラムとリンクできるオブジェクトモジュールを集めたものです。リンカはまずプログラムのオブジェクト入力ファイルをリンクします。外部シンボルが未定義の場合（つまり、それらがどのオブジェクト入力ファイルにも現れない場合）、リンカはライブラリファイル内でシンボルを捜します。あるライブラリのオブジェクトモジュールの中に未定義のシンボルが見つかった場合、そのオブジェクトモジュールはプログラムにリンクされます。リンカは、他のモジュールに参照される外部シンボルの定義を持つこれらのオブジェクトモジュールを、ライブラリからインクルードするだけです。ライブラリは通常、標準の実行時関数を保持するために高級言語と共に使われます。たとえば、C言語実行時ライブラリは、"printf"、"strcpy"、"abs"などの関数を持ちます。

例：　386 LINK SIEVE -LIB　¥MSC¥SLIBC -8086

省略形：-LIBの代わりに-L

3.9　-MAPスイッチと-NOMAPスイッチ

構文：　-MAP file
-NOMAP

説明：　-MAPスイッチにより、リンカによって生成されるマップファイルの名前と位置の指定ができます。通常、リンカは、最初のオブジェクト入力ファイルと同じ名前のマップファイルを作成します。ただし、ファイル名拡張子は".MAP"に変わります。マップファイルは最初のオブジェクト入力ファイルと同じディスクおよびディレクトリに置かれます。このデフォルトマップファイル名が受け入れられない場合には、その名前を変えるのに"-MAP"スイッチを使います。MAPファイルが不要な場合"-NOMAP"スイッチを指定します。リンカはマップファイルを作成せずにユーザプログラムをリンクします。

例：　386 LINK MAIN -MAP MYMAP
　　　（デフォルト拡張子は".MAP"となる）
386LINK A B C -MAP A:MYMAP
　　　（ディスクAにマップを送る）
386LINK NEWDOS -MAP ¥JUNK¥NEWDOS
　　　（ディレクトリ ¥JUNKにマップを送る）
386LINK HELLO -NOMAP
　　　（リンカはマップファイルを作成しない）

省略形：-MAPの代わりに-M
-NOMAPの代わりに-NOM

3.10 -MINDATAスイッチと-MAXDATAスイッチ

構文：　-MINDATA number
-MAXDATA number

説明：　MS-DOSは、.EXEファイルをロードするとき、.EXEファイルを保持するのに十分な大きさのメモリブロックを割り当てる必要があります。MS-DOSはオプションとして、プログラムの終りのあとに、そのプログラムが使える追加メモリを割り当てます。-MINDATAスイッチと-MAXDATAスイッチは、.EXEファイルを形成するのに使われます。これによりMS-DOSは、追加のデータを何バイト割り当てるかがわかります。

-MINDATAスイッチはMS-DOSが割り当てる追加データの最小バイト数を決定します。バイトカウントは、数値定数として-MINDATAの後ろに与えられます。特に指定がなければ、リンカはその数値を10進数とみなします。異なる基数を指定するために、接尾文字がその数値に付加されます。使用できる接尾文字を次に示します。

接尾文字	基数
B	2進数
H	16進数
Q	8進数

-MAXDATAスイッチは、プログラムに割り当てられる追加バイトの最大数を指定するのに使われます。-MINDATAのように、最大バイトカウントはスイッチの後ろに与えられます。

コマンドラインに-MINDATAスイッチが与えられない場合は、リンカはゼロ（0）バイトとみなします。これは、プログラムが確実に納まり、使用できるバイトが余分にある場合に、メモリにプログラムがロードできることを意味しています。-MAXDATAスイッチの指定がない場合、リンカはすべてのメモリが利用できるものとみなします。これは、プログラムのロード時、すべての利用可能メモリがプログラムに割り当てられることを意味しています。
注：最小値および最大値は、.EXEファイルに置かれる前に16バイトパラグラフ境界にまで丸められます。

例：　386 LINK SPECIAL -MINDATA 100H -MAXDATA 100H

省略形：-MINDATAの代わりに-MIND
-MAXDATAの代わりに-MAXD

3.11 -NOOUTPUT スイッチ

構文：　-NOOUTPUT

説明：　リンカは通常、リンクされたプログラムを格納した.EXEファイルを生成します。未定義シンボルなどのエラーを見つけるだけの目的でプログラムをリンクする場合は、"-NOOUTPUT"スイッチを指定してリンカに出力ファイルを作成しないように指示できます。リンカはプログラムをリンクし、プログラム内にエラーがあることを知らせますが、不要な.EXEファイルは生成しません。エラーを修正した後、きれいにリンクされたプログラムファイルを得るために"-NOOUTPUT"を取り除くことができます。

例：　386 LINK BUGGY BUGS1 BUGS2 -NOOUTPUT

省略形：-NOOUTPUT の代わりに-NOO

3.12 -ONECASEスイッチと-TWOCASEスイッチ

構文：　-ONECASE
-TWOCASE

説明：　パブリックシンボル、セグメント、およびグループの名前を処理する場合、リンカは名前を形成する英文字の大文字、小文字を区別します。リンカは、大文字、小文字を別々の文字とみなします。たとえば、名前 "abc"と "ABC"とは異なります。しかしながら、コンパイラによっては名前の大小文字は区別せず、"abc" も"ABC" も同じ文字とみなすものもあります。そこで、オプションとして、ユーザのコンパイラと互換性を持たせるように、リンカが名前の大文字、小文字を区別しないようにできます。スイッチ"-ONECASE"は、名前の大小文字の区別を無視することをリンカに伝えるのに使われます。スイッチ "-TWOCASE" は、大小文字を区別するよう指定するのに使用します。リンカのデフォルト操作では大小文字を区別するので"-TWOCASE"スイッチを使用する必要はありません。

例：　386 LINK MAIN SUB1 SUB2 -ONECASE

省略形：-ONECASEの代わりに-OC
-TWOCASEの代わりに-TC

3.13 -STACKスイッチ

構文：　-STACK size

説明：　-STACKスイッチは、プログラムのスタックセグメントのサイズを指定するのに使用されます。このスイッチのあとには、スタックに割り当てられるサイズ（単位：バイト）を指定しなければなりません。サイズは、386 | ASM によって許容されるのと同じフォーマットの数値定数として指定されます。特に指定のない場合、リンカはこの数値を10進数とみなします。異なる基数を指定するには、接尾文字がこの数値に付加されます。使用できる接尾文字を以下に示します。

接尾文字	基数
B	2進数
H	16進数
Q	8進数

たとえば、4Kバイトのスタックを割り当てるには、"-STACK 4096" あるいは "-STACK 1000H " を使用できます。

スタックセグメントがすでにプログラムに存在している場合、このセグメントのサイズを変更するためにスイッチ"-STACK"が使われます。スタックセグメントは、そのセグメントが定義されるとき、組合せ型のSTACK を指定することで作成されます。-STACKスイッチは、プログラムによって割り当てられたサイズをオーバーライドします。しかしながら、リンカはそのスタックセグメントのサイズを大きくすることができるだけです。スタックセグメントのサイズを小さくしようとすると、エラーになります。

プログラムにスタックセグメントが存在しない場合は、リンカはプログラムの最終セグメントのあとにセグメントを作成します。そのセグメントのサイズは
-STACKスイッチによって指定されたサイズになります。そのセグメント名は "STACK" になります。

プログラムにスタックセグメントの定義がされておらず、コマンドラインに -STACKスイッチが与えられていない場合は、リンカはスタックセグメントが存在しない旨の警告メッセージを与えます。.EXEファイルを正しく実行するには、スタックセグメントが必要です。

例：

```
386LINK HELLO -STACK 1024
386LINK HELLO -STACK 200H
```

省略形：-STACKの代わりに-S

付録A コマンドスイッチの要約

コマンドライン（オプション）を以下に示します。

386LINK file1,file2,...	入力ファイル（.OBJ）
-LIB file1,file2,...	ライブラリファイル（.LIB）
-EXE file	.EXE出力ファイル（.EXE）
-HEX file	インテル16進出力ファイル
-NOOUTPUT	出力ファイルなし
-MAP file	マップファイル（.MAP）
-NOMAP	マップファイルなし
-DUMP	オブジェクトファイルをダンプする。
-MINDATA size	最小データサイズを設定する。
-MAXDATA size	最大データサイズを設定する。
-STACK size	スタックサイズを設定する。
-ONECASE または -TWOCASE	シンボルの大文字、小文字を選択する。
-8086 、-80286、または-80386	ターゲットCPU を選択する。

付録B エラーメッセージ

B.1 エラー

リンカに回復可能なエラーが発生した場合、 386 | LINKは単純エラーであることを通知します。単純エラーが発生すると 386 | LINKは画面に"Error:"を表示します。この表示のあとにはそのエラー内容を説明するメッセージが表示されます。これと同じメッセージがMAPファイルにも書き込まれます。

本節では 386 | LINKで起こりうる単純エラーをすべて説明します。また、そのエラーについて考えられる原因と処置も説明します。エラーメッセージは、参照しやすいようにアルファベット順に示します。

Error: Duplicate definition of the symbol "XXX" in module "YYY".

原因： 同一シンボルを2回以上定義しようとした。

処置： 余分なシンボル定義を削除するか、あるいは二つのシンボル内の一つのシンボルの名前をリネーム（名前変更）する。

Error: Fix-up overflow in segment "XXX" at offset "YYY". Value is "ZZZ".

原因： DSレジジスタまたはCSレジジスタが示すアドレスから65,535バイト以上離れている変数またはプロシジャを参照しようとした。

処置： その変数またはプロシジャを、DSレジスタまたはCSレジスタでアドレス指定できるように適切なセグメントに移動させる。

Error: Segment "XXX" has overflowed.

原因： 8086または80286 のプログラムをリンクする場合、一つのセグメントに65,535バイトを超えるデータを置こうとした。

処置： オーバーフローしたセグメントを2個以上の、より小さいセグメントに分割する。

B.2 フェータルエラー

リンカに続行不能なエラーが発生すると、 386 | LINKはフェータルエラーを表示します。フェータルエラーが発生すると、 386 | LINKは "Fatal Error:" を画面に表示します。この表示のあとにはそのエラー内容を説明するメッセージが表示されます。フェータルエラーの発生した状態に関係なく、.EXEファイルまたは.HEXファイルは処分され、マップファイルが残されます。このときのマップファイルは空か、一部だけ書き込まれた状態です。

本節では、 386 | LINKで発生するすべてのフェータルエラーをその原因と考えられる処置と共に説明します。エラーメッセージは、参照しやすいようにアルファベット順に示します。

Fatal Error:　Bad object file...

原因：　本付録の§3．を参照してください。

Fatal Error:　Cannot create linker output file --
　　　　　　　Access denied to file "XXX".
Fatal Error:　Cannot create map file -- Access denied to file "XXX".
Fatal Error:　Cannot open object input file -- Access
　　　denied to file "XXX".

原因：　1．誤ったディスク名が指定された。
　　　　2．別のユーザがファイルをロックしている。
処置：　アクセス問題を解決する。

Fatal Error:　Cannot create linker output file -- No such file or
　　　　　　　directory "XXX".
Fatal Error:　Cannot create map file -- No such file or directory "XXX".
Fatal Error:　Cannot open object input file -- No such file or
　　　　　　　directory "XXX".

原因：　1．ファイル名、あるいはそのファイルが常駐しているディレクトリ名にスペルミスがある。
　　　　2．当該ファイル、あるいはそのファイルが常駐していると思われるディレクトリがない。
処置：　そのファイル名またはディレクトリ名のスペルを正しく綴り直すか、そのファイルまたはそのディレクトリを新しく作成する。

Fatal Error: Cannot create linker output file -- Disk full "XXX".
Fatal Error: Cannot create map file -- Disk full "XXX".

原因： 指定されたファイルの置かれているディスクが一杯でこれ以上ファイルを持つことができない。
処置： 同一ディスクに常駐している未使用ファイルを処分するか、別ディスクにそのファイルを記憶させる。

Fatal Error: Cannot create linker output file -- Too many files open.
Fatal Error: Cannot create map file -- Too many files open.
Fatal Error: Cannot open object input file -- Too many files open.

原因： リンカが、DOS の許容を超えた数のファイルをオープンしようとした。
処置： CONFIG.SYSファイルの FILESパラメータを、8だけ増やす。DOS のマニュアルにCONFIG.SYSの変更方法が説明されている。CONFIG.SYSの変更後は、この変更を有効にするためにシステムをリブートしなければならない。

Fatal Error: Cannot open command file -- "XXX".

原因： リンカはコマンドファイルを見つけることができなかった。
処置： コマンドファイル名のスペルを修正するか、そのコマンドファイルを新たに作成する。

Fatal Error: Command too big -- "XXX".

原因： コマンドファイル内で、コマンドスイッチ、スイッチの値、あるいはファイル名が 255文字を超えている。
処置： その文字列を短くする。

Fatal Error: Error reading from command file -- "XXX".

原因： リンカがコマンドファイルを読み込めなかった。その理由はおそらくファイルがディスク上で損傷しているためである。
処置： コマンドファイルを再作成する。

Fatal Error: Extra command line argument -- "XXX".

原因： オブジェクト入力ファイル名がコマンドスイッチの後に与えられた。入力ファイル名はすべてコマンドスイッチの前に与えなければならない。

処置：　オブジェクト入力ファイル名をスイッチの前に移動させる。

Fatal Error:　Invalid file name -- "XXX".

原因：　正しいファイル名構文に従わないファイル名がコマンドライン上に与えられた。たとえば、

386LINK...XYZ

処置：　ファイル名を修正する。

Fatal Error:　Missing value to command switch -- "XXX".

原因：　コマンドスイッチは指定されたが、そのスイッチに必要な値が与えられなかった。たとえば、

386LINK HELLO -MAP

処置：　スイッチに値を指定する。

Fatal Error:　No object input files were specified to be linked.

原因：　リンクするオブジェクトファイルがリンカに与えられなかった。たとえば、

386LINK -EXE MYPROG -MAP MYPROG

処置：　一つ以上のオブジェクト入力ファイルを正しく指定する。

Fatal Error:　No other arguments are allowed with a command file name.

原因：　コマンドファイル名を持つコマンドラインに、他のコマンドスイッチが与えられた。たとえば、

386LINK @MYLINK -8086

処置：　すべてのコマンドスイッチをコマンドファイルに移動する。

Fatal Error:　Out of memory.

原因：　リンカがメモリ不足になった。

処置：　より多くのメモリを持つシステム上でリンクする。現バージョンの386|LINKは、リンクされる全プログラムをメモリ内に保持する。プログラムが、利用可能なメモリより大きい場合、そのプログラムはリンクできない。

Fatal Error:　Unknown Command Switch -- "XXX".

原因：　リンカにとって理解できないコマンドスイッチがリンカに与えられた。たとえば、

386LINK HELLO -NOEXE

処置：　その無効なコマンドスイッチのスペルを修正する。

B.3　オブジェクトファイルエラー

386 | LINKは、リンクするすべてのオブジェクトファイルとライブラリファイルのフォーマットをチェックします。無効なフォーマットを持つオブジェクトファイルに出会うと、リンカはフォータルエラーを通知し、実行を停止します。リンカは "Fatal Error: Bad object file" を画面に表示します。このメッセージの後には、オブジェクトファイルの何が間違っているかを説明するメッセージが続いて表示されます。不良オブジェクトファイルエラーの原因は通常、そのファイルを生成したアセンブラまたはコンパイラにあります。

アセンブラまたはコンパイラが不良オブジェクトファイルを生成する一般的な理由をいくつか次に示します。

- オリジナルのソースプログラムにエラーがあった。
- アセンブラまたはコンパイラにバグがある。
- アセンブラまたはコンパイラがインテルOMF-86規格に従っていない。

オブジェクトファイルエラーメッセージが現れたら、まずソースファイルを再コンパイルするか再アセンブルして、オブジェクトファイルを構築し直します。コンパイラまたはアセンブラが何らかのエラーを通知したら、再リンクする前にこれらのエラーを修正します。アセンブリまたはコンパイルにエラーがなくなったら、リンクをもう一度試みてください。

386 | LINKで発生するオブジェクトファイルエラーを以下に示します。

Fatal Error:　Bad object file -- bad checksum in "XXX".

原因：　あるレコードのチェックサムバイトに間違った値があり、レコードのすべてのバイトの総和がゼロ（0）にならない。

Fatal Error:　Bad object file -- bad end record on "XXX".

原因：　オブジェクトファイルに、不適正な値を持つエンドレコードがある。

Fatal Error:　Bad object file -- bad explicit fix-up entry in "XXX".

原因：　フィックスアップレコード内の明示的フィックスアップエントリが、違法な値を持っている。

Fatal Error:　Bad object file -- bad library file type in "XXX".

原因：　ライブラリファイル以外のファイルあるいは正しいフォーマットではないファイルをリンクしようとした。386 | LINKは、マイクロソフト社のLIB ユーティリティ、またはLIB と互換性のあるライブラリユーティリティによって作成されるライブラリファイルをリンクできる。

Fatal Error:　Bad object file -- bad segment class name index in "XXX".

原因：　セグメント定義時に、クラスインデックスがセグメントクラスではない名前を参照している。

Fatal Error:　Bad object file -- fix-up record has no data record in "XXX".

原因：　フィックスアップレコードが参照できるデータレコードがない。このエラーは、フィックスアップレコードがデータレコードのあとではなく、前に出力された場合に発生する可能性がある。

Fatal Error:　Bad object file -- illegal name index in "XXX".

原因：　インデックス値を使用して論理名を参照しようとしたが、その名前がLNAMESレコードにまだ定義されていない。

Fatal Error:　Bad object file -- illegal record type in "XXX".

原因：　レコードの最初のバイトにレコードタイプが指定されている。このエラーは、そのタイプバイトに有効なレコードタイプ番号がない場合に発生する。

Fatal Error:　Bad object file -- improper record length in "XXX".

原因：　オブジェクトレコードが短かすぎてそのレコードのすべてのフィールドを記憶するとができないか、そのレコードのチェックサムバイトの前に余分なバイトがある。

Fatal Error:　Bad object file -- logical name defined more than once.

原因：　論理名を２回以上定義しようとした。

Fatal Error:　Bad object file -- logical name defined more than once in "XXX".

原因：　単一のグループまたはセグメントを２回以上定義しようとした。

Fatal Error:　Bad object file -- logical name used before defined in "XXX".

原因：　論理名をそれが定義される前に使おうとした。

Fatal Error: Bad object file -- Unexpected EOF

原因： リンカがオブジェクトファイルから読み込もうとしたが、ファイルにそれ以上のデータがないことが分かった。オブジェクトファイルを作成したプログラムがファイルの書き込みを終了しなかったか、ファイルの最終レコードとしてエンドレコードを書くのに失敗した。

Fatal Error: Bad object file -- unknown file format in "XXX".

原因： オブジェクトファイル以外のファイル、あるいはOMF-86規格に準拠してないファイルをリンクしようとした。

Fatal Error: Bad object file -- zero repeat count in iteractive data record in "XXX".

原因： 繰返しデータレコードのリピートカウントがゼロである。リピートカウントは正の値でなければならない。

付録C オブジェクトモジュールフォーマット

386 | LINKがオブジェクトファイル（".OBJ"ファイル）に使用するフォーマットは、"簡易 OMF-386"と呼ばれます。簡易 OMF-386は、インテル社とマイクロソフト社が使用しているOMF-86の単純な拡張です。本付録では簡易 OMF-386とOMF-86の相違について説明します。OMF-86については、参考文献を参照してください。

アセンブラまたはコンパイラは、次のコメントレコードをオブジェクトモジュールの先頭に置くことより、オブジェクトモジュールのターゲットが80386 であることをリンカに知らせます。

0	1	2	3　　　　7	8
88H	80H	AAH	" 80386 "	
レコードタイプ	フラグ	クラス		チェックサム

80386 コメントレコードは、モジュールヘッダレコード（THEDER）の直後で、オブジェクトモジュールの他のレコードより前に置く必要があります。

オブジェクトモジュールの他のレコードは、8086と同じように簡易 OMF-386にフォーマットされます。しかし、8086と異なる点は、オブジェクトレコードのオフセット、変位、セグメント長の各フィールドの長さが2バイトではなく4バイトになっていることです。

次のレコードは、サイズが増加するフィールドを持っています。

レコード：	フィールド：
SEGDER	オフセットおよびセグメント長
PUBDEF	オフセット
LEDATA	オフセット
LIDATA	オフセット
Explicit FIXUP	ターゲット変位
BLKDEF	復帰アドレスオフセット
LINNUM	オフセット
MODEND	ターゲット変位

フィックスアップレコードには、次の二つの”Loc ”値が新たに定義されています。

5－　32ビットオフセット

6－　ベース＋32ビットオフセット（ロングポインタ）

参考文献

1. Intel Corp., 8086 Relocatable Object Module Formats(Order Numbar 121748-001). 1981
2. Steven Armbrust and Ted Forgeron ".OBJ Lessons," PC Tech Journal, October 1985, Vol.3, No.10, p.62.

付録D .EXEファイルフォーマット

.EXEファイルは以下のヘッダで始まります。

オフセット（16進数）	内容（1ワード）
00	サイン（' MZ' ）
02	ファイルサイズを512で割った剰余
04	ファイルサイズを512で割った商
06	再配置項目数
08	ヘッダサイズ　（単位：パラグラフ）
0A	最小データ　（単位：パラグラフ）
0C	最大データ　（単位：パラグラフ）
0E	初期SS
10	初期SP
12	チェックサム
14	初期IP
16	初期CS
18	先頭の再配置項目のオフセット
1A	オーバーレイの数

合計 1C

オフセット	内容
0 0	.EXEファイルであることを示すファイルサイン。サインは文字"M"と"Z"である。
0 2	ファイルサイズを512 で割った剰余（単位：バイト）。
0 4	ファイルサイズを512 で割った商（単位：バイト）。
0 6	ファイル内の再配置項目の数。
0 8	ヘッダと再配置項目のサイズの和（単位：16バイトパラグラフ）。この数は.EXEファイル内のプログラムの開始位置を見つけるのに使われる。
0 A	プログラムのメモリロード時に、プログラムの終端に割り当てられる16バイトパラグラフの最小数。
0 C	プログラムのメモリロード時に、プログラムの終端に割り当てられる16バイトパラグラフの最大数。16進数"FFFF"は、利用できるすべてのメモリを割り当てることを意味する。
0 E	このワードとプログラムがロードされるセグメントとの総和が、スタックセグメントレジスタSSの初期化に使われる。
1 0	スタックポインタレジスタSPの初期値。
1 2	ファイルのチェックサム。チェックサムはファイルのすべてのワードを加算し、けた上げを無視し、その和に否定演算を行うことにより、計算される。
1 4	命令ポインタレジスタIPの初期値。
1 6	このワードと、プログラムがロードされるセグメントとの総和が、コードセグメントレジスタCSの初期化に使われる。
1 8	最初の再配置項目のファイルの始めからのオフセット（単位：バイト）。
1 A	オーバーレイ番号、プログラム主要部の場合はゼロ。

再配置項目はヘッダの次に置きます。各再配置項目は4バイト長です。一つの再配置項目には、フィックスアップが必要なプログラム内の、セグメント：オフセットフォーマットのアドレスが含まれています。再配置項目の最初のワードはオフセットであり、2番目のワードはセグメントです。メモリロケーションは1ワード長であり、セグメントアドレスを含んでいます。メモリロケーションは、プログラムがロードされるセグメントアドレスをそのロケーションに加算して、ロード時にフィックスアップます。

付録E インテル16進オブジェクトファイルフォーマット

386｜LINKのオプションとして、リンクされたプログラムはインテル16進オブジェクトファイルの形式で出力することができます。インテル16進オブジェクトファイルフォーマットは、ほとんどのPROMプログラムおよびCPU エミュレータの入力ファイルとして使用されます。本付録ではインテル16進オブジェクトファイルフォーマットについて説明します。

インテルＨＥＸオブジェクトファイルは文字としてコード化されたレコードから構成されます。レコードは以下の基本フォーマットに従います。

0	1－2	3－6	7－8	9－	
:	レコード長	ロードアドレス	レコードタイプ	データ ～	チェックサム

バイト0　　　レコードの開始を表すASCII 文字 ":"

バイト1～2　　レコードのデータバイト数を与える16進数。最大値は16進の "FF"

バイト3～6　　メモリに配置されるレコードのデータバイトのロードアドレス（データレコードにのみ有効。このフィールドは、他のレコードタイプでは"0000" となる）。

バイト7～8　　2桁の16進数で与えられるレコードタイプ。有効な値を次に示す。

"00"　データレコード
"01"　ファイルレコードの終り
"02"　拡張アドレスレコード
"03"　開始アドレスレコード

バイト9～n　　このフィールドは可変長である。このフィールドは、0またはそれ以上の数の16進数字対としてコード化されたデータバイトからなる。

最終2バイト　　レコードの最終2バイトは8bitチェックサムである。チェックサムは、"："とチェックサムフィールド自体を除く、レコードのすべての　フィールドのバイト単位の総和に否定演算を行ったものである。チェックサムは16進2桁としてコード化される。

リンカは、80文字を超えるレコードの出力はしません。リンカは、各レコードの後にキャリッジリターン／ラインフィードを組み合わせて出力します。16進数字は、数字"0"～"9"および"A"～"F"のASCII 文字で表されます。

拡張アドレスレコード

0	1 ～ 2	3 ～ 6	7 ～ 8	9 ～12	13 ～14
:	'02'	'0000'	'02'	セグメントアドレス	チェックサム

拡張アドレスレコードは、以降のすべてのデータレコードをロードするセグメントを指定するのに使われます。レコードのデータフィールドには16進4桁としてコード化された16ビットのセグメントアドレスが含まれています。セグメントアドレスが指定されていない場合は、ゼロとみなされます。

データレコード

0	1 ～ 2	3 ～ 6	7 ～ 8	9 ～	
:	レコード長	ロードアドレス	'00'	データ	チェックサム

データレコードには、メモリにロードされるデータバイトが含まれています。ロードアドレスフィールドは、データバイトをメモリにロードする場合の16ビットオフセットを与えます。このオフセットフィールドは、前の拡張アドレスレコードによって指定されるセグメントアドレスと組み合わされて、ロードアドレス全体を作成します。ロードされるデータバイトには、データレコード内のバイトオフセット9で始まる複数組の16進数字が与えられます。データレコードのレコード長フィールドは、存在するデータバイト数を指定します。

開始アドレスレコード

0	1〜2	3〜6	7〜8	9〜12	13〜16	17〜18
:	'04'	'0000'	'03'	CS	IP	ﾁｪｯｸｻﾑ

開始アドレスレコードは、ファイルのロード後に実行される、ファイルの最初の命令のアドレスを指定します。開始アドレスは、セグメント：オフセットフォーマットで与えられます。CSフィールドにはセグメントアドレスが含まれており、IPフィールドには開始アドレスのオフセットが含まれています。CSフィールドとIPフィールドは16進で４桁で表現される16ビットの数字です。

ファイルレコードの終り

0	1〜2	3〜6	7〜8	9〜10
:	'00'	'0000'	'01'	ﾁｪｯｸｻﾑ

ファイルレコードの終りは、オブジェクトファイルの終りをマークします。このレコードは、常に16進オブジェクトファイルの最終レコードです。

付録F 使用例

1. ファイルHELLO.OBJ をLINKして、HELLO.EXE とHELLO.MAP とを生成する。

```
386LINK HELLO
```

2. プログラムHELLO をサブルーチンモジュールSUB1、SUB2およびSUB3とリンクする。

```
386LINK HELLO SUB1 SUB2 SUB3
```

3. プログラムHELLO をサブルーチンライブラリCLIBとリンクする。

```
386LINK HELLO -LIB CLIB
```

4. 8086用のプログラムHELLO をリンクする。

```
386LINK HELLO -8086
```

5. リンクをコマンドファイルから実行する。

```
386LINK @LINKIT
```

6. マップファイルを抑止する。

```
386LINK TEST -NOMAP
```

7. .EXEファイルを抑止する。

```
386LINK TEST -NOOUTPUT
```

8.　.EXEファイルと.MAPファイルをリネーム（名前変更）する。

```
386LINK CHDR TEST -LIB CLIB -EXE TEST -MAP TEST
```

9.　オブジェクトファイルをダンプするためにリンカを使用する。

```
386LINK TEST -NOOUTPUT -DUMP -MAP TEST.DMP
```

索引

386｜ASM TOOL KIT ユーザーズマニュアル
第2部 リンカ 386｜LINK

英数・記号

あ行

さ行

た行

な行

ま行

ら行

第3部

シンボリックデバッガ

386｜DEBUG

目次

386 | ASM TOOL KIT ユーザーズマニュアル
第3部 シンボリックデバッガ 386 | DEBUG

第1章 概要

386｜DEBUG は、Intel 80386 マイクロプロセッサ用のシンボリック機械語デバッガで、386 ネイティブモードまたはリアルモードのどちらかで実行するように作成されたプログラムをデバッグするのに使用します。

386｜DEBUG は、386｜DOS-Extenderによって提供されるのと同じ386 ネイティブモードプログラム実行環境を提供します。また、リアルモードプログラムは標準MS-DOS環境でデバッグされます。

386｜DEBUG コマンド名と一般形式は、MS-DOSで提供されるSYMDEBユーティリティにあるコマンド名および一般形式と同じです。 386｜DEBUG では、386 ネイティブモードとリアルモードの両モードにおいて80386 の全レジスタの表示と変更が可能であり、また、80386 の全命令を逆アセンブルすることができます。さらにメモリの表示、変更、移動、比較を行う命令や、プログラム中にブレイクポイント、ウォッチポイントをセットする命令およびシングルステップをセットする命令が追加されています。

第2章 386 | DEBUG の使用

2.1 コマンドライン構文

デバッガを実行するのに使用するコマンドラインには、デバッガEXE ファイル名（386 DEBUG)と、そのあとにデバッガコマンドラインスイッチ、デバッグするプログラム名そしてアプリケーションプログラムに受渡す引数が順に並びます。

```
386debug <debug switches> <program name> <arguments>
```

デバッガコマンドラインの例を以下に示します。

```
386debug myprog
386debug -maxreal 200h myprog
386debug pecho This is a protected mode program
```

コマンドラインスイッチは、デバッガのデフォルトの動作を変更するのに使用します。デバッグするプログラム名は、フルネームで指定することも、ファイル拡張子を付けずに指定することもできます。ファイル拡張子の指定がない場合、386 | DEBUGは、まず拡張子".EXP"（386 ネイティブモードプログラムのデフォルトのファイル名拡張子）の付いたファイルを探します。".EXP"ファイルがない場合は、".EXE"（リアルモードプログラムのデフォルトのファイル名拡張子）の付いたファイルを探します。

386 | DEBUG は、EXP 形式（386 | LINKが386ネイティブモードプログラム用に作成するデフォルトのファイル形式）、EXE 形式（386 | LINKがリアルモードプログラム用に作成するデフォルトのファイル形式）、および REX形式（386 | LINKが作成する別の386ネイティブモードプログラム形式）の三つのプログラムファイル形式のデバッグをサポートします。さらに、386 | DEBUG は、Phar Lap社の386 | DOS-Extenderとバインドされた 386ネイティブモードプログラムのデバッグにも使用できます。プログラムファイル形式の詳細は、『第2部　リンク』を参照してください。

2.2 コマンドラインスイッチ

コマンドラインスイッチは、 386 | DEBUG のデフォルトの動作を変更するのに使用されます。デフォルトでは、 386 | DEBUG は以下の仕様になっています。

・スモールメモリモデルプログラムがデバッグされているとみなす。
・プログラムシンボルの検索時、大文字／小文字の区別をしない。
・デバッガコマンド入力をシステムキーボードから読み込み、デバッガ出力をシステム画面に表示する。

コマンドラインスイッチは、マイナス符号("-")で始まり、その後にスイッチ名が続きます。各々のスイッチ名には完全形と短縮形の２形式があります。引数がある場合は、スイッチ名の後ろに区切り記号として１スペースをあけて引数を指定します。一つのコマンドラインに矛盾する複数のスイッチが指定された場合、最後（右端）のスイッチが優先します。

デバッガスイッチによっては、引数として数値を必要とするものがあります。デフォルトでは、この数値は10進数（基数10）とみなされます。

16進数を指定するには、数値の後に"h" または"H" をつけます。次の二つの例は、スイッチ"-MAXREAL"の引数として同じ数値を指定していることになります。

```
386debug -maxreal 200h hello
386debug -maxreal 512 hello
```

2.2.1 ケースセンシティビティスイッチ

-ONECASEスイッチと-TWOCASEスイッチは、 386 | DEBUG が大文字／小文字を区別するかどうかを指定するのに使用されます。デフォルトでは、デバッガは、すべてのプログラムシンボルの大文字／小文字を区別しません。たとえばデフォルトでは、"my_sym"、"my_SYM"、"MY_SYM"は、すべて同じシンボルとみなされます。-TWOCASEスイッチは、デバッガが大文字／小文字の区別をするよう指定し、-ONECASEスイッチは、大文字／小文字の区別を無効にする場合に使用します。-ONECASEスイッチはデフォルト指定と同じことです。

-TWOCASEスイッチは、 386 | LINKのプログラムリンク時に-TWOCASEスイッチを使用した場合に限り、デバッグの際に使用します。

構文： -ONECASE
　　　 -TWOCASE
省略形：-ONEC
　　　 -TWOC
例：　 386debug -twocase hello

2.2.2　省略時ポインタサイズスイッチ

-SMALLスイッチ、-COMPACTスイッチ、-MEDIUM スイッチおよび-LARGEスイッチは、デバッグ中のプログラムのメモリモデルを指定するのに使用します。これは、KとKA(stack trace)コマンドおよびポインタ間接式演算子"*"を正しく動作させるのに必要なものです。-SMALLスイッチはスモールメモリモデルプログラム（NEARコードとNEARデータポインタ）を、-COMPACTスイッチはコンパクトメモリモデルプログラム（NEARコードとFAR データポインタ）を、-MEDIUM スイッチはミディアムメモリモデルプログラム(FARコードとNEARデータポインタ）を、また-LARGEスイッチはラージメモリモデルプログラム（FARコードとFARデータポインタ）をそれぞれ指定します。プログラムのメモリモデルは、コンパイル時に決定されます。

ポインタサイズスイッチの指定がない場合、デフォルトの設定はスモールメモリモデルプログラムとなります。また、386 ネイティブモードプログラムのデバッグ時には、スモールメモリモデルを必ず選択する必要があります。現バージョンの 386|LINKでは、386ネイティブモードプログラムに対してはスモールメモリモデルでなければならないからです。

構文：　-SMALL
　　　　-COMPACT
　　　　-MEDIUM
　　　　-LARGE

省略形：-SM
　　　　-COMP
　　　　-MED
　　　　-LG

例：　-386debug -medium hello
　　　-386debug -large hello

2.2.3　386|DOS-Extenderスイッチ

386|DEBUG は、386|DOS-Extenderでサポートされるコマンドラインスイッチもすべて認識します。しかし、プログラムは、これらのコマンドスイッチを必要としません。以下に 386|DOS-Extenderスイッチの概要を示します。各コマンドスイッチの詳細は、『386|DOS-Extenderユーザーズマニュアル』を参照してください。

-MINREAL n	最低限必要なリアルモード空きメモリ
-MAXREAL n	リアルモード空きメモリの最大量
-REALBREAK n	リアルモードメモリにロードするアプリケーションプログラム量

-CALLBUFS n	プロシジャコールを386 ネイティブモードとリアルモード間で行う場合に、データ受渡し用としてアプリケーションプログラムで利用するモード間プロシジャコールバッファに割り当てられるメモリ量
-MINBUF n	DOS システムコール時に使用するデータバッファに割り当てる最小メモリ量
-MAXBUF n	DOS システムコール時に使用するデータバッファに割り当てる最大メモリ量
-NISTACK n	割り当てる割込みスタックバッファの数。この数は、実行される386 ネイティブモードとリアルモード間の入れ子にされた切り換えの最大数を制御する
-ISTKSIZE n	各割込みスタックバッファのサイズ
-HWIVEC n	ハードウェア割込みIRQ0-IRQ7 を再配置する割込みベクタベース
-EXTLOW n	アプリケーションプログラムがHIGHメモリ中で使用可能な領域の最小アドレス
-EXTHIGH n	アプリケーションプログラムがHIGHメモリ中で使用可能な領域の最大アドレス

2.3 386 | DEBUG の起動

デバッグされるプログラムには、そのプログラム内のすべてのパブリックシンボルを収めたシンボルテーブルが組み込まれていなければなりません。シンボルテーブルを組み込むには、プログラムを 386 | LINKでリンクするときに、コマンドラインで-SYMBOL スイッチを使用します。シンボルテーブルまたは.EXPファイルか.EXEファイルの終りに付加され、386 | DEBUG によって自動的にロードされます。

プログラムのデバッグを 386 | DEBUG の制御下で開始するには、2.1 節で述べたようにコマンドラインを以下のように入力します。

```
C>386debug hello
```

386 | DEBUG は、プログラムをメモリにロードし、そのあとに次のようなメッセージを表示します。

```
386 | DEBUG:1.1--Copyright(C)1986,1987 Phar Lap Sofrware Inc.
 [2000 bytes in the symbol table]

 [80386 protected mode]
```

メッセージの表示後、 386 | DEBUG はマイナス符号("-") を表示します。この符号は、最初のデバッガコマンドを入力できる状態であることを示す。 386 | DEBUG は、一つのコマンドが終了し次のコマンドの入力準備ができるたびにこの符号を表示します。

メッセージの１行目は、実行中の 386 | DEBUG のバージョン番号、２行目は、プログラムのシンボルテーブルのサイズを示します（単位：バイト）。４行目は、アプリケーションがリアルモード、386 ネイティブモードのうちのどちらのモードで実行されているかです。リアルモードアプリケーションは、80386 CPU を8086/88 CPU のように使用します。386 ネイティブモードでは、アプリケーションは80386 CPU の全機能を使用できます。

2.4　コマンド構文

386 | DEBUG のコマンドは、すべて、１文字か２文字のコマンド名とそのあとに続くコマンドオペランドから構成されます。複数のオペランドは、スペースまたはタブで区切ります。本書では、省略可能なオペランドは、括弧“［　］”でくくります。これらのオペランドの指定がなかった場合は、デフォルト値を指定したものとみなされます。各コマンドのデフォルトのオペランドはコマンドの説明の中で説明します。

2.4.1　コマンドオペランド

386 | DEBUG コマンドのオペランドは、以下の基本要素から構成されます。

- そのコマンドが使用するパブリックシンボル名
- 80386 レジスタの名前。このレジスタ名は、コマンド（たとえば、change register コマンド）によって操作されるレジスタを指定する場合とそのレジスタ内容を値として使用する（たとえば、データセグメントのEBX が指すメモリを表示する）場合とがあります。これはコマンドの文脈によります。
- 絶対数。 386 | DEBUG はすべての絶対数を16進数と解釈します。
- 単一引用符でくくった文字列（'ABC' など）
- “:” 文字。この文字は、アドレスをセグメント：オフセット形式（たとえば、FS:100）で入力するのに使用されます。
- 記号“+”と“−”。これらの記号は、簡単な式中で絶対数、シンボルおよびレジスタ値を結合するのに使用されます。
- 記号“*”。この記号は、間接演算子として式中で使用されます。
- 括弧“(”と“)”。式をくくるのに使用されます。

例：	-R EAX	レジスタEAX の内容を表示する。
	-DB 100	DS:100のメモリを表示する。
	-U main	関数“main”を逆アセンブルする。
	-DD ptr_array	“ptr_array” の内容を表示する。
	-DB *bufptr	変数“bufptr”が示すメモリを表示する。
	-DB FS:EBX	FS: ［EBX ］のメモリを表示する。
	-F 0 50 '<*>'	文字列値でメモリを埋める。
	-DB EBX+50	DS: ［EBX+50］のメモリを表示する。

2.4.2　シンボル

デバッグセッションの開始時、デバッガは、.EXPファイルか.EXEファイルの終りにあるシンボルテーブルをメモリに読み込みます。このシンボルテーブルにはプログラム中の各パブリックシンボルのエントリが入っています。アセンブリ語プログラムでは、シンボルはPUBLICディレクティブによってパブリックシンボルと宣言されます。Cプログラムの場合は、すべての関数とスタティック、宣言されていないグローバル変数がすべてパブリックシンボルとなります。

各々のシンボルには、そのメモリ内のアドレスと同じ値が割り当てられます（シンボル値）。シンボルをコマンド中で使用するたびに、デバッガは、シンボルをシンボル値に置換します。したがって、シンボルは、デバッガコマンド中メモリアドレスが置かれるところならどこでも使用できますし、式中で組み合わせることもできます。Xコマンドは、すべてのパブリックシンボルとその値をリストします。またこのコマンドは、特定のシンボル値を求めるのにも使用できます。

デフォルトでは、386|DEBUGは、シンボル検索中大文字／小文字の区別をしません。プログラム構築のために大小文字区別リンクを実行した（-TWOCASEスイッチが386|LINKと一緒に使われた）場合、シンボル検索において大小文字区別をするため、386|DEBUGと一緒に-TWOCASEスイッチを使用する必要があります。

リアルモードプログラム中のシンボル値は、完全なセグメント：オフセット形式のアドレスです。オフセットは、プログラムリンク時に決定され、セグメントパラグラフアドレスは、プログラムのメモリへのロード時に決定されます。386 ネイティブモードプログラム中のシンボル値は、386|LINKによって作られた単一プログラムセグメント中のオフセットだけです。

絶対数はすべて、16進数で入力されるので、386|DEBUGは、英字Ａ－Ｆを数字の一部とみなします。したがって、386|DEBUGは｛０－９，Ａ－Ｆ｝以外の文字が１文字も含まれていないシンボルをシンボルとして認識しません。たとえば、プログラムがパブリックシンボル“ABC”を持っている場合、386|DEBUGは、このシンボルを16進数字と解釈し、シンボルテーブルを検索しないので、このシンボルを参照することはできないわけです。

例：　-U chk_err　　エラーチェックルーチンを逆アセンブルする

　　　-DB buf+40バッファ“buf”の最初の64バイトの次のアドレスからメモリ内容(128バイト）を表示する

2.4.3　式

コマンドオペランドは、(1)シンボル名、(2)絶対数、(3)その内容が値として使用されるレジスタと(4)シンボル名、絶対数、レジスタ内容を組み合わせて386 | DEBUG によって一つの値が求められる式で構成されます。

式中の演算はすべて、32ビット符号なし数字として扱われる値で実行されます。

式演算子は４種類あり、そのうち加算演算子（+）と減算演算子（-）だけが、算術演算子です。間接演算子（*）は、１レベルの間接アドレス指定を行うのに使用されます。セグメントオーバライド演算子（:）は、セグメント：オフセット形式のアドレス中でセグメントセレクタ値をオフセットと分けるのに使用されます。演算子の優先順位は以下の通りです。演算子は、優先順位の高いものが低いものより先に評価され、同じ順位の演算子では、左から右へと評価されます。

演算子		優先順位
*	（間接アドレス）	最上位
+ -		↓
:	（セグメントオーバライド）	最下位

式中に括弧を使用すると、演算子の優先順位を変えることができます。括弧内の演算は隣接する演算より常に先に評価されます。

例：

```
-R EAX                    ──→ EAXの値を表示する。
EAX=00000050
-R EBX                    ──→ EBXの値を表示する。
EBX=00000100
-DB EBX                   ──→ DS:00000100のメモリを表示する。
-DB EBX-EAX               ──→ DS:000000B0のメモリを表示する。
-DB CS:20+90-5            ──→ CS:000000ABのメモリを表示する。
-X bufptr                 ──→ シンボル"bufptr"をリストする。
bufptr= 00000400
-DD bufptr bufptr+4       ──→ "bufptr" にあるポインタを表示する。
0014:00000400  00000800
-DB *bufptr               ──→ "bufptr" が指すメモリ(DS:00000800) を表示す
                              る。
-DB *bufptr+100           ──→ DS:00000900のメモリを表示する。
-DD ES:0 4
004C:00000000  00000A00
-DB CS:*(ES:0)            ──→ CS:00000A00のメモリを表示する。
```

2.4.4 アドレス

386 | DEBUG コマンドの多くは、メモリのアドレスをコマンドオペランドとして必要とします。メモリアドレスは、セグメントセレクタ値とそのセグメント内のバイトオフセットで指定されます。 386ネイティブモードでは、セグメントセレクタは、セグメントを記述するLDT かGDT 中のエントリへのインデックスです。リアルモードでは、セグメントセレクタはメモリのセグメントのパラグラフベースアドレスです。

386 | DEBUG アドレスオペランドは、単純なオフセットでも完全なセグメント：オフセット形式のアドレスでも指定できます。オフセットは、16進数で指定されるか、または 386ネイティブモードプログラムのデバッグ中はプログラムシンボルを使って指定されます。式にオフセットだけしか指定されない場合、コマンドに従ってデフォルトのセグメントが指定されます。U (unassemble) 、G (go) 、BP (set breakpoint) 、T (trace)、TQ (trace quality)、P (procedure trace)、およびPQ (Ptrace quietly) のコマンドは、すべて、CSレジスタの示すセグメントをデフォルトのセグメントとみなし、他のコマンドはすべてDSレジスタが示すセグメントをデフォルト値とみなします。

完全なセグメント：オフセット形式のアドレスは、式中で以下のいずれかの方法によって指定できます。

1. リアルモードプログラムをデバッグする場合、プログラムシンボルを使うと、そのシンボルが定義されているセグメントを指定できます。
2. 80386 レジスタの内容を式中に使用した場合、デフォルト時のセグメントは、80386 プロセッサの想定するデフォルト時のセグメントと同じです。つまり、ESP、EBP、SPまたはBPのレジスタ内容で形成されたアドレスは、SSセグメントを想定し、EIP またはIPのレジスタ内容で形成されたアドレスはCSセグメントを想定します。その他のレジスタ内容で形成されたアドレスはすべてDSセグメントを想定します。
3. セグメントセレクタ値は、セグメントオーバライド演算子":" を使って指定できます。
4. 間接演算子"*" が式中に使用されているとき、FAR ポインタがメモリから検出される場合は、結果のアドレス値にはセグメントセレクタが含まれます。NEARポインタの検出時は、アドレス値はオフセットだけです。

なお、式演算子と、その演算子をメモリアドレスの作成に使用する方法は、2.4.6 項で詳しく説明します。

例：		
	-DB 100	DS:100のメモリを表示する。
	-G =100	GoをCS:100で実行する。
	-DB EBX	DS: [EBX] のメモリを表示する。
	-DB ESP	SS: [ESP] （スタックのトップ）のメモリを表示する。
	-U *ESP	スタックのトップのプロシジャリターンアドレスで逆アセンブルする。

-G =fatal-err	ルーチン"fatal-err" でGoを実行する。
-DB buf	バッファ"buf" の内容を表示する。
-DB buf+EBX	"buf" の最初のEBX バイトの次のアドレスからメモリ内容（128 バイト）を表示する。
-DB FS:EBP+20	FS: [EBX+20] のメモリを表示する。
-DB 3C:00	003C:00000000 のメモリを表示する。

2.4.5　領域

アドレス領域内のメモリに作用するコマンドはいくつかあります。アドレス領域は、開始アドレスと終了アドレスを指定することによって定義されます。コマンドの多くが領域を必要としますが、領域の定義は開始アドレスの指定だけでかまいません。領域の終りについてはデフォルトの終了アドレス（通常、開始アドレスのあとの特定のバイト数）が想定されます。また、領域の開始アドレスも指定しなくてもよい場合も多くあります。その場合、コマンドはデフォルト値（通常、最後にコマンドを使用した時の指定領域の終了アドレスのすぐあとのバイトか、コマンドの使用が全くなかった場合にはゼロ）を使用します。本書では、オペランドとしてアドレス領域をとるコマンドの説明には、以下の表記法を使用します。

range	開始アドレスと終了アドレスの両方を指定する必要がある場合
address | range	開始アドレスは必須で、終了アドレスは省略可能の場合
[range]	開始アドレスと終了アドレスとも省略可能の場合

アドレス領域を指定する場合、開始アドレスに対してだけ完全なセグメント：オフセットアドレスが指定されます。領域の終了アドレスは、開始アドレスと同一セグメントになければならないので、終了アドレスに対するセグメントオーバライドはできません。

例：	-DB buf buf+100	"buf" の最初の256 バイトを表示する。
	-DB 100 200	DS:100からDS:200までのメモリ内容を表示する。
	-DB CS:10 20	CS:10 からCS:20 までのメモリ内容を表示する。
	-DB 0	DS:0からDS:7F までのメモリ内容（128 バイト）を表示する。
	-DB	DS:80 （最後に使用したDBコマンドの終了アドレスのすぐあとのアドレス）からDS:FF までのメモリ内容を表示する。

2.4.6 式演算子

(1) 加減算演算子

演算子"+"と"-"は、2個のオペランドの加算または減算を実行します。メモリアドレスにおいては、ほとんどの場合、一つのオペランドだけが完全なセグメント：オフセット形式のアドレスで、他のオペランドは単純オフセット（絶対数）でなければなりません。唯一の例外は、シンボルをレジスタの内容と組み合わせる場合です。この場合は、指定されたセグメントセレクタが異なることがあり、シンボルが定義されているセグメントがレジスタ用に想定されたデフォルトのセグメントより優先します。

例：	-DB 50+70-10	DS:B0 のメモリ内容を表示する。
	-T 3+5	8つの命令をトレースする。
	-DB EBP+10	SS:10 [EBP] のメモリを表示する。
	-DB buf+BP	"buf"の最初のBPバイトの次のアドレスからメモリ内容（128 バイト）を表示する。

(2) セグメントオーバライド演算子

セグメントオーバライド演算子":"は、メモリアドレスのセグメントセレクタ部をオーバライドするのに使用されます。":"の左側のセグメントセレクタ値は、右側の式によって指定されたセグメントセレクタを置き換えます。この演算子は、一つのコマンドに対して 386 | DEBUG が指定したデフォルトのセグメントが望みのセグメントでない場合に役立ちます。

例：	-DB CS:20	コードセグメント内のメモリを表示する。
	-DB FS:EBP	FS: [EBP] のメモリを表示する。
	-G =4C:10	実行を004C:10 から開始する。

(3) 間接演算子

間接演算子(*)は、メモリに格納されたポインタを介して1レベルの間接アドレスを行うのに使用します。この演算子を使用すると、メモリに格納されたポインタは、そのポインタの示すメモリ位置に直接アクセスできます。つまり、2個のコマンド（ポインタをダンプするコマンドと、このコマンドから得た16進のポインタ値を使うコマンド）を入力する必要がなくなるわけです。

"*"演算子は、NEAR（オフセットのみ）かFAR（ セグメント：オフセット）のポインタのうちどちらを選択するかを決めるのに、プログラムのメモリモデルを使用します。プログラムのメモリモデルは、-SMALL、-COMPACT、-MEDIUM または-LARGEのコマンドラインスイッチか、MS, MC, MMまたはMLのデバッガコマンドで指定します。

データとコードに別タイプのポインタを使用するメモリモデル（COMPACT,MEDIUMなど）の場合、“*”演算子は、コードポインタとデータポインタのどちらかを選択します。これは、コマンドのデフォルトのセグメントがコードセグメントであるか、データセグメントであるかに基づきます。たとえば、コマンドが U(unassemble)の場合、“*” 演算子は、コードポインタを選択し、DB(display memory)コマンドの場合には、“*” 演算子は、データポインタを選択します。各コマンドのデフォルトのセグメントは、2.4.4 項で説明しています。

“*” 演算子のオペランドは、ポインタが格納されているメモリアドレスです。通常、ポインタのアドレスは、完全なセグメント: オフセットか、オフセットのみかどちらかの形式で指定します。オフセットだけの指定がある場合、デバッガコマンドのデフォルトのセグメントがポインタを格納しているセグメントを知るのに使用されます。

メモリから検出したポインタアドレスは、そのポインタを検出するのに使ったアドレスを完全に置き換えます。NEARポインタが検出された場合、結果のアドレスは、オフセット値だけを指定します。この指定は、ポインタを格納しているメモリアドレスがセグメントセレクタを指定するかどうかには関係なく行われます。

例：	-DB *bufptr	変数 “bufptr” の指示するメモリを表示する。
	-S *bufp *bufp+400 FF	“bufp”の示す配列を検索して、値FFを持つバイトを見つける。
	-U *(EBP+4)	カレントルーチンからのリターンアドレスの命令を逆アセンブルする。
	-DD FS:10 14	
		NEARポインタ値を表示する。
	0054:00000010 000002A0	
	-DB *(FS:10)	DS:2A0のメモリを表示する。
	-DB CS:*(FS:10)	CS:2A0のメモリを表示する。

2.5　386 | DEBUG ディスプレイ出力の制御

386 | DEBUG のコマンドには、膨大な量の表示出力（たとえば、巨大なメモリブロックの表示）を生成するコマンドがいくつかあります。これが発生した場合、(1)希望する情報が上方スクロールされて画面から消えるのを防ぐためにディスプレイへの出力を一時的に停止したり、(2)ディスプレイへの出力を取消して、デバッガコマンドプロンプトに戻すことが可能です。

CTRL/S文字（CTRLキーを押したままS キーを押す）は、ディスプレイへの出力を一時的に停止するのに使用されます。画面上の情報を見た後、任意のキーを押せば、デバッガはディスプレイへの出力を再開します。

CTRL/Cは、一つのコマンドの残りのディスプレイ出力部分を取消し、デバッガコマンドプロンプトに戻るのに使用されます。

2.6 入出力リダイレクション

通常、デバッガコマンドはキーボードから入力され、デバッガ出力はディスプレイに表示されます。しかし、デバッガコマンドをディスクのスクリプトファイルから読み込み、デバッガ出力をディスクのログファイルにリダイレクトすることも可能です。

コマンド入力のリダイレクトは、記号“<”と、そのあとにコマンドが登録されたファイル名とをコマンドラインに入力して指定します。入力ファイルの空白行は無視されます。また、コメント行（注釈などを入れておくための行で、実行されない）であることを示すには、行の先頭に特殊文字“*”を置きます。コマンドの処理は、Q(quit)コマンドにぶつかるか、ファイルが終了するまで続けられます。Qコマンドがファイルにない場合、デバッガは、ファイルの終りに達したときにコマンド処理をやめます。

デバッガ出力のリダイレクトは、“>”と、そのあとに出力を書き込むファイル名とをコマンドライン上に入力して指定します。

入力のリダイレクトなしに、出力だけをリダイレクトすることもできます。しかし、ちょっと注意が必要です。というのは、出力がリダイレクトされるときは、画面に表示される文字すべてが出力ファイルに書き込まれますが、これには、通常、画面にエコーで返されるキーストロークも含まれます。したがって、入力のリダイレクトなしに出力だけがリダイレクトされている場合は、何がキーボードから入力されたかを見ることができないのです。

例：　386debug hello
コマンドをキーボードから入力し、デバッガ出力をディスプレイに表示する。
386debug <script hello
“script”ファイルからコマンドを読み込み、デバッガ出力をディスプレイに表示する。

386debug <script >log hello
“script”ファイルからコマンドを読み込み、デバッガ出力を“log”ファイルに書込む。

2.7 制御の獲得

386 | DEBUG コマンドは、デバッガに制御権があり、コマンドプロンプト"-" を表示しているときだけ入力できます。アプリケーションプログラムの実行時、デバッガに制御を戻す方法はいろいろあります。 386 | DEBUG は制御を取り戻すたびに、その理由とすべての80386 レジスタの内容を表示します。

2.7.1 プログラムブレイクポイントと命令トレース

プログラムのブレイクポイントにくると、 386 | DEBUG は指定されたブレイクポイントアドレスの命令を実行する直前に、アプリケーションの実行を停止します。 386 | DEBUG に制御が移ると、80386 レジスタの内容とコマンド入力のためのプロンプトが表示されます。ブレイクポイントでは、メモリとレジスタの内容を調べてエラーの原因を究明するために、エラーのありそうな地点までプログラムを実行させることができます。

ブレイクポイントには、テンポラリー（一時的）ブレイクポイントとパーマネント（常置）ブレイクポイントの２種類があります。テンポラリーブレイクポイントはgo(G) コマンドで設定され、 386 | DEBUG が制御を取り戻すと、その理由にかかわらず、必ず消去されます。テンポラリーブレイクポイントを有効なままにしておくには、Ｇコマンドの入力のたびに指定する必要があります。パーマネントブレイクポイントは、BPコマンドで設定されます。このブレイクポイントはBCコマンドで消去されるかBDコマンドで無効にされるまで有効です。パーマネントブレイクポイントは、たとえば、何回も呼び出される重要な関数にブレイクポイントを設定するのに役に立ちます。テンポラリーブレイクポイントはいちいち消去する必要がないので、プログラムを段階的に進める場合に役立ちます。

ブレイクポイントは、命令の境界に設定する必要があります。命令の途中に設定されたブレイクポイントは認識されないので、デバッガに制御が戻りません。特定の命令のアドレスを調査するには、逆アセンブル(U) コマンドを使用します。

ブレイクポイントに関係するコマンドとして、T(trace)コマンドがあります。このコマンドは、テスト中のプログラムの命令を一つ実行し、そのあとに制御をデバッガに戻します。このコマンドは、Ｇコマンドを使って次の命令にブレイクポイントを設定するのと同じ効果があります。この機能は、バグがありそうな一連の命令を段階的に実行する場合に役立ちます。

例：	-BP printf	パーマネントブレイクポイントをプリントルーチンにセットする。
	-G fatal_err	テンポラリーブレイクポイントをフォータルエラールーチンにセットする。
	-G 55 7B	テンポラリーブレイクポイントをCS:55 とCS:7B にセットする。
	-T 5	五つの命令をトレースする。

2.7.2　データウォッチポイント

データウォッチポイントを指定すると、 386 | DEBUG は、特定のメモリロケーションを読書きする命令の実行直後にアプリケーションの実行を停止します。データウォッチポイントにくると、デバッガはアクセスされたメモリロケーションと現在のレジスタの内容を表示した後、コマンド入力のためのプロンプトを表示します。データウォッチポイントは、プログラム内のコードまたはデータをくずしてしまうバグを見つける場合に役立ちます。

ウォッチポイントは、WPコマンドで設定します。ウォッチポイントの設定時、ウォッチポイントのアドレスと予想されるアクセスのサイズ（1バイト、2バイトまたは4バイト）の両方を指定する必要があります。データサイズが正しくないと、そのメモリロケーションへのアクセスをしません。さらに、ウォッチポイントは、書込みアクセスだけか、読出しと書込みの両方のアクセスに設定するかを指定することができます。デフォルトでは、WPコマンドは書込みだけのウォッチポイントを設定します。読出し書込みウォッチポイントを設定するには、文字"r" をWPコマンドの最後のパラメータとして加えます。

例：	-WP bufp 4	ポインタ変数"bufp"の変更時だけに、ウォッチポイントが発生するように設定する。
	-WP errf 2 r	エラーフラグの読出し時か書込み時にウォッチポイントが発生するように設定する。
	-WP main+20 1	"main"ルーチンのプログラムがワイプアウトされるときにウォッチポイントが発生するように設定する。

2.7.3　プロセッサ例外

80386 プロセッサは、いくつかのプログラムエラーを検出することができます。これらのエラーが発生すると、80386 CPU はプロセッサ例外を発生させます。割込み番号00から20（16進）までは、プロセッサ例外用として確保されています。 386 | DEBUG は、すべての80386 プロセッサ例外のハンドラをインストールしているので、例外が発生すると必ず制御を獲得します。80386 プロセッサ例外を以下に示します。

番号（16進）	エラー
00	ゼロで除算する
01	デバッガトレースとブレイクポイント
02	マスク不能外部割込み
03	INT3デバッガブレイクポイント
04	オーバフロー（INTO命令）
05	BOUND 命令（配列バウンドチェック）
06	命令opコード違反（通常、プログラムがデータ内へジャンプすることによって発生する）
07	演算コプロセッサが使用できない
08	ダブルフォールト

09	コプロセッサセグメントオーバラン（コプロセッサがセグメントの終りを超えてデータをアクセスしようとした）
0A	TSS 無効（タスクスイッチに使用するタスク状態セグメントが無効）
0B	セグメント不在（GDT またはLDT にないセグメント記述子を使用しようとした）
0C	スタックフォールト（SSセグメントレジスタを使用する操作でのセグメント制限違反）
0D	一般的な保護規則違反
0E	ページフォールト（ページが存在しない）

2.7.4 プログラムの終了

テスト中のプログラムが終了すると、制御は当然デバッガに戻ります。デバッガは、DOS exitコール発生場所と、コール時のレジスタ値を認識していないので、レジスタ内容は表示されません（デバッガは、DOS プログラム終了コードが実行されるまで制御権がありません）。代わりに、プログラムによってDOS に受渡されたexit値が表示されます。

プログラム終了後、G(go) コマンド、T(trace)コマンドおよびP(procedure trace)コマンド以外ならどんなコマンドでも実行できます。この３コマンドが実行できないのは、プログラムに割り当てられたメモリがすでに解放されているため、有効な命令またはスタックスペースがないからです。

2.7.5 CTRL/C

DOS コールを行うプログラムへの割込みは、CTRL/Cを入力します。DOS はCTRL/Cをキーボードバッファ中に認識すると制御を獲得し、割込み番号23（16進）を出します。この割込み番号は、デバッガによって処理されます。この機能は、無限ループに陥ったプログラムに割込む場合に役立ちます。

リアルモードプログラムと386 ネイティブモードプログラムとのデバッグでは、CTRL/Cに対する動作に少し違いがあります。

386 ネイティブモードでのプログラム実行中に（結果としてCTRL/C割込みを招く）DOSコールを出した場合、レジスタ表示はDOS コール発行時のレジスタ内容を表示します。CTRL/Cの入力で制御を386 ネイティブモードプログラムから得た後、 386|DEBUG は、レジスタの内容の変更をすべて禁止します。これは、アプリケーションプログラムが再び制御を得る前にDOS コールは処理を完了しているからです。ただし、ブレイクポイントとT(trace)コマンドは自由に使用できます。

CTRL/Cを入力してリアルモードプログラムに割込むと、レジスタ表示には、DOS がCTRL/C割込みを出す時のレジスタの内容を表示します。この時デバッガは、レジスタ内容の変更を許可しますが、変更はしないほうがよいでしょう。変更すると、割込みされたシステムコールの処理をDOS がどう終了するかに影響してくるからです。T(trace)コマンドは、この場合 386 | DEBUG にアプリケーションプログラムの命令ではなく、DOS の命令をトレースさせます（386 ネイティブモードプログラムの場合と異なります）。

2.8 386ネイティブモードと リアルモードプログラムとのデバッグの違い

386 | DEBUG は、リアルモードプログラムと386 ネイティブモードプログラムの両方をデバッグできます。リアルモードプログラムは、デフォルトではファイル拡張子".EXE"を持ち、386 ネイティブモードプログラムは、デフォルトではファイル拡張子".EXP"を持ちます。 386 | DEBUG はどららのモードでプログラムを実行するかをメッセージで表示します。

386 | DEBUG は、どちらのモードでデバッグする場合も、同一の基本コマンドを提供します。リアルモードプログラムのデバッグ時に利用できないコマンドは、386 ネイティブモードシステムテーブル表示用コマンド（DG, DL, DI, PD, PT）だけです。他のコマンドはすべてどちらのモードでも全く同じ動作をします。

リアルモードと386 ネイティブモードとのデバッグ上の大きな違いは、セグメントセレクタに使用される値にあります。386 ネイティブモードプログラムの場合、セグメントセレクタは、LDT(ローカル記述子テーブル、DLコマンドで表示できる）のインデックスで、小さな値になります（たとえば、コードセグメントに対して16進数"0C"、データセグメントに対して16進数"14"）。リアルモードプログラムの場合には、セグメントセレクタ値は、セグメントベースのパラグラフアドレスであるため、16進数"3D69"などの大きな値になります。

また、386 ネイティブモードプログラムの場合、シンボル値は、CSとDSの示す単一プログラムのセグメント内の32ビットオフセットです。一方、リアルモードプログラムの場合は、セグメント：オフセット形式（セグメント値は、シンボルの位置するセグメントメモリ内の16ビットのパラグラフベースアドレスであり、オフセット値はセグメント内の16ビットオフセット）となります。

386 ネイティブモードコマンド例：

DB buf	バッファ"buf" 内のデータを表示する。
DB CS:100	CSの示すセグメントのオフセット100 にあるデータを表示する。
DB C:100	"0C"のセレクタ値を持つセグメントのオフセット100 にあるデータを表示する。

リアルモードコマンド例：

DB buf	バッファ"buf" 内のデータを表示する。
DB CS:100	CSの示すセグメントのオフセット100 にあるデータを表示する。
DB 3D69:100	3D69のセレクタ値（パラグラフアドレス）を持つセグメントのオフセット100 にあるデータを表示する。

第3章 デバッガコマンドの説明

本章では、各 386 | DEBUG コマンドについて、以下の事項を説明します。

・コマンドの構文。オプションの（省略可能な）オペランドは、大かっこ（[ ]）で囲んで表示する。
・コマンドの説明
・コマンドの使用例

3.1 BC、BD、BE、BL、BPコマンド

\- Breakpoint Control（ブレイクポイントの制御）

構文：

```
BC breakpts
BC *
BD breakpts
BD *
BE breakpts
BE *
BL
BP address
```

説明：　ブレイクポイント制御コマンドは、ブレイクポイントの設定、消去、有効化、無効化およびリストに使用されます。ブレイクポイントは、命令の境界に設定しなければなりません。ブレイクポイントをセットすると、デバッガは、その命令の実行前に制御を獲得します。

BPコマンドは、ブレイクポイントをセットするのに使用します。コマンドオペランドは、ブレイクポイントをセットする命令のアドレスです。BPコマンドでセットされるブレイクポイントは、有効（オン）にも無効（オフ）にもできます。ブレイクポイントは、BPコマンドで初めてセットされたときは必ず有効になっています。

BLコマンドは、BPコマンドでセットしたブレイクポイントをリストします。各ブレイクポイントは、 386|DEBUG によって割り当てられた番号で識別されます。BLコマンドは、個々のブレイクポイントに対してそのブレイクポイント番号、ブレイクポイントをセットした命令のアドレス、および 386|DEBUG がブレイクポイントの設定に用いた内部メカニズム（80386 ハードウェアレジスタの１つを使用した場合は、"Debug reg"、１バイトデバッグ割込み命令を使用した場合は、"INT3"）を表示します。加えて、ブレイクポイントが無効になっている場合には、BLコマンドは、ブレイクポイント情報の後に、"***disabled***"を表示します。
BDコマンドはブレイクポイントを一時的に無効にするのに使用され、BEコマンドはBDコマンドで無効にしたブレイクポイントを有効にするのに使用されます。これらのコマンドは、ブレイクポイント番号のリストか、特殊文字"*"（すべてのブレイクポイントを意味する短縮形）のどちらかを引数としてとります。

BCコマンドは、ブレイクポイントをクリアするのに使用します。このコマンドは、ブレイクポイント番号のリストか特殊文字"*"（すべてのブレイクポイントを意味する短縮形）を引数としてとります。ブレイクポイントはいったんクリアされると、BLコマンドでリストすることも、BEコマンドで有効にすることもできなくなります。クリア後再びブレイクポイントが必要になった場合は、BPコマンドで再設定する必要があります。

例：

```
-BP main
-BP err_chk
-BP main+9
-BL
0 MAIN mechanism=Debug reg
1 ERR_CHK  mechanism=Debug reg
2 MAIN+09 mechanism=Debug reg
-BD 0 2
-BL
0 MAIN mechanism=Debug reg ***disabled***
1 ERR_CHK  mechanism=Debug reg
2 MAIN+09 mechanism=Debug reg ***disabled***
-BE *
-BC 1
-BL
0 MAIN mechanism=Debug reg
2 MAIN+09 mechanism=Debug reg
```

3.2　C コマンド

－　Compare Memory （メモリの比較）

構文：　C range address

説明：　C コマンドは、２個のメモリブロックを比較して、等しいかどうかを判断します。メモリブロック間に相違点があれば、画面に表示されます。C コマンドは三つのオペランドを持ちます。最初の二つのオペランドには、比較する最初のメモリブロックのアドレス領域を指定します。最後のオペランドは、２番目のメモリブロックの開始アドレスです。２番目のブロックの終了アドレスは指定する必要はありません。デバッガは、常に２番目のブロックのサイズを最初のブロックと同じとみなします。両ブロックのどちらにもセグメントセレクタが指定されないときは、デフォルト値として、DSレジスタの値が使われます。

例：　-C 100 1FF 200

メモリロケーション100 から1FF までと200 から2FF までをバイトごとに比較する。対応するバイトが異なる場合、デバッガは各バイトのアドレスと値を次の形式で表示する。

```
      0014:00000115 FF A5 0014:00000215
               ↑      ↑  ↑         ↑
ブロック1 のアドレス ─┘  │  └─ ブロック2 のアドレス
  ブロック1 の内容 ──────┘     └── ブロック2 の内容
```

-C buf1 buf1+100 buf2

バッファ“buf1”と“buf2”の先頭の256 バイトを比較する。

3.3　D 、DA、DB、DD、DWコマンド

－　Display Memory（メモリの表示）

構文：　D　[range]

DA [range]

DB [range]

DD [range]

DW [range]

説明：　メモリ表示コマンドは、メモリロケーションを画面に表示するのに使用します。これらのコマンドは、表示するメモリブロックを指定する領域オペランド（オプション）を持ちます。開始アドレスの指定がない場合、最後に使用された表示コマンドで表示された領域の終了のバイトの次のバイトから表示されます。

終了アドレスの指定がない場合、128 バイトのメモリ領域が表示されます。
DBコマンドはメモリロケーションをバイト形式で表示します。データバイトは、16進とASCII の両形式で表示され、印字不可能なASCII 文字はピリオド(".")として表示されます。

DWコマンドは、メモリロケーションを16進ワード（16ビット値）形式で表示します。

DDコマンドは、メモリロケーションを16進ダブルワード（32ビット値）形式で表示します。

DAコマンドは、データをASCII 形式でだけ表示します。このコマンドは、ゼロバイトデータを見つけるまで、または（ゼロバイトがない場合）指定された領域の終了ポイントまでデータを表示します。

D コマンドは、最後に使用された表示コマンドを短縮形にしたものです。たとえば、DDコマンドの後にD コマンドを指定した場合、メモリロケーションはダブルワード形式で表示されます。

アドレスでセグメントが指定されなかった場合、メモリ表示コマンドは、DSの示すセグメントをデフォルト値と想定します。

例：

```
-DB buf buf+F
0014:00000100 45 72 72 6F 72 00 01 02-12 64 00 00 00 00 00 00 Error...d.


-DW buf buf+F
0014:00000100 7245 6F72 0072 0201 6412 0000 0000 0000

-DD buf buf+F
0014:00000100 6F727245 02010072 00006412 00000000

-DA buf buf+F
0014:00000100 Error

-DW CS:0 F
000C:00000000 139A 0000 0005 0200 A5A5 0090 001A 0000
```

3.4 DG、DI、DLコマンド

- Display System Tables（システムテーブルの表示）

構文：　DG [range]
DI [range]
DL [range]

説明：　DGコマンド、DIコマンドおよびDLコマンドは、それぞれシステムGDT（グローバルディスクリプタテーブル）、IDT（割込みディスクリプタテーブル）およびLDT（ローカルディスクリプタテーブル）を表示します。各コマンドは、表示するテーブルエントリの開始インデックスと終了インデックスとの二つのオペランド（オプション）をとります。インデックス番号0はテーブルの最初のエントリを指定し、1は2番目の欄を指定します。

オペランドの指定がない場合、16個のテーブルエントリが表示されます。このエントリの表示は、前にコマンドを使用したときに最後に表示されたエントリの次から始まります。開始インデックスだけ指定した場合、指定インデックスのエントリから16のエントリが表示されます。開始インデックスと終了インデックスの両方が指定された場合、開始インデックスから終了インデックスまでのすべてのエントリが表示されます。

GDT は、386 | DEBUG の使用するセグメントをマッピングします。LDT は、アプリケーションプログラムの使用するセグメントをマッピッグします。特定セグメント割当ての詳細は、『386 | DOS-Extenderユーザーズマニュアル』を参照してください。DGコマンドまたはDLコマンドは、GDT またはLDT のエントリごとに以下の情報を表示します。値は、すべて16進で表示されます。

- セグメントのアクセスに使用するセグメントセレクタ
- セグメントリニアベースアドレス
- セグメント制限（バイト制限または4KBページ制限のどちらかであることに注意）
- セグメントディスクリプタのフラグバイト（アクセス権バイト）
- セグメント使用型 ………… 16ビットセグメントにはUSE16 、32ビットセグメントにはUSE32
- セグメント制限の細分性 …· バイト細分またはページ細分

セグメントディスクリプタ（フラグバイトのビット定義を含む)の詳細は、80386プログラマ用文法書を参照してください。IDT には、80386 のサポートする256 個の各々の割込みに対して割込み処理ルーチンを示すディスクリプタが含まれている。DI コマンドは、IDT のエントリごとに以下の情報を表示します。値は、すべて16進で表示されます。

- IDT のエントリに対応する割込み番号（テーブル内のエントリのインデックスと同じ）
- 割込み処理ルーチンのセグメントセレクタとオフセット
- ディスクリプタのフラグバイト

なお、DGコマンド、DIコマンド、DLコマンドは、リアルモードプログラムのデバッグ時は使用できません。

例：
```
-DG 0 1
#0000 segment not present
#0008 base=00025900 limit=0000FFFF flags=9B USE32 byte granularity

-DL 0 1
#0004 base=00044D60 limit=000000FF flags=92 USE32 byte granularity
#000C base=00400000 limit=00000345 flags=9A USE32 page granularity

-DI FF
#00FF selector=0008 offset=0000AFF5 flags=8E
```

3.5 E 、EB、ED、EWコマンド

-Enter (change) Memory（メモリの入力（変更））

構文：
```
E  address value(s)
EB address value(s)
ED address value(s)
EW address value(s)
```

説明：　メモリ入力コマンドは、メモリロケーションを変更するのに使用します。E コマンドとEBコマンドは、同義であり、メモリを1バイトずつ変更します。EWコマンドはメモリを1ワードずつ変更し、EDコマンドは、メモリをダブルワード単位で変更します。

メモリ入力コマンドのオペランドは、変更するメモリのアドレスとその後に置く入力用のリスト(一つ以上の値)から構成されます。アドレスと一緒にセグメントセレクタが指定されなかった場合、DSの示すセグメントがデフォルト値として想定されます。

E コマンドとEBコマンドは、メモリ入力用値として16進値のほかに文字列も受入れます。文字列中の文字は、別々のバイトとしてメモリに格納されます。

例：
```
-DB buf buf+F
0014:00000100 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ················

-EW buf 1234
-DB buf buf+F
0014:00000100 34 12 00 00 00 00 00 00-00 00 00 00 00 00 00 00 4 ···············

-EB buf FF 'HELLO' FF
-DB buf buf+F
0014:00000100 FF 48 45 4C 4C 4F FF 00-00 00 00 00 00 00 00 00 .HELLO·········

-ED 200 12345678
-DB 200 20F
0014:00000200 78 56 34 12 00 00 00 00-00 00 00 00 00 00 00 00 xV4 ·············
```

3.6 F コマンド

- Fill Memory（メモリの充てん）

構文： F range value(s)

説明： Fコマンドは、1ブロックのメモリを特定値に初期化するのに使用されます。Fコマンドのオペランドは、メモリ領域とそのあとに続く一つ以上の充てん用のデータバイトから構成されます。領域は、充てんされるメモリのロケーションを指定します。充てん用の値は、バイト値か文字列のうちのどちらかでなくてはなりません。また、この値は、指定領域の充てんに必要な回数だけ繰返されます。アドレスと一緒にセグメントが指定されなかった場合、DSの示すセグメントがデフォルト値として想定されます。

-F 100 1FF 0
DS:100から1FF までのメモリロケーションをすべてゼロにセットする。

-F 1000 1200 'HELLO'
DS:1000 から1200までのメモリロケーションを文字列"HELLO"を何度も繰返して充てんする。

3.7 G コマンド

- Go

構文：　G [=address] [address]

説明：　G コマンドは、プログラムの実行を開始するのに使用されます。最も単純な形式のgoコマンドは、オペランドをとりません。このコマンドは、CS:EIP（命令ポインタ）の示す命令から実行を開始します。2.7 節で述べたいずれかの条件が発生すると制御はデバッガに戻ります。

G コマンドは、制御をプログラム内の任意の位置に移すのにも使用できます。制御を移す開始アドレスは、アドレスの前に等号(=) をつけることで示されます。

G コマンドは、さらにテンポラリーブレイクポイントをプログラム内に設定することもできます。テンポラリーブレイクポイントを設定する命令のアドレスをコンドオペランドとしてリストします。ブレイクポイントは、1度に最大4個まで指定できます。すべてのテンポラリーブレイクポイントは、386 | DEBUG が制御を取戻すとクリアされます。テンポラリーブレイクポイントを設定のままにしておきたい場合、次のG コマンドでそのブレイクポイントを再入力する必要があります。

ブレイクポイントアドレスか、開始アドレスでセグメントセレクタが指定されない場合、CSレジスタのセグメントセレクタがデフォルト値として使用されます。

例：　-G

現在のCS:EIPから実行を続行する。

-G rd_line

テンポラリーブレイクポイントを関数 “rd_line” に設定する

-G =C:20

000C:20 から続行する。

-G =fatal_err printf exit

実行を"fatal_err"から続行し、ブレイクポイントを “printf” と “exit” に設定する。

3.8 H コマンド

- Hex Arithmetic （16進演算）

構文:　H value value

説明：H コマンドは、16進計算器としてデバッガ中で使用されます。このコマンドは、二つの32ビットの16進数字をオペランドとしてとります。H コマンドは、その二つの数値の合計と差を計算し、結果を画面に16進数で表示します。

例：

```
-H AAAAA 12345
000AAAAA + 00012345 = 000BCDEF
000AAAAA - 00012345 = 00098765
```

3.9 I コマンド

- Input from Port（ポートからの入力）

構文：I port

説明：I コマンドは、入力ポートからバイト値を読み取るのに使用します。このコマンドのオペランドは、読み取る入力ポートの番号だけです。デバッガは、読み取った値を、16進数で表示します。

例：

```
-I 0
Port 0000 = 00
```

3.10 K 、KAコマンド

- Stack Trace（スタックトレース）

構文：K [arg_count]
KA

説明：スタックトレースコマンドは現在のプロシジャコールスタックを表示するのに使用されます。表示の１行目には現プロシジャとプロシジャ内のオフセットが表示されます。２行目以降には、プロシジャコールを最高レベルのプロシジャまでトレースバックし、１行に１プロシジャコールずつ表示します。たとえば、２行目には、現プロシジャを呼び出したプロシジャと、現プロシジャの復帰時に実行される命令のアドレスとが表示されます。

スタックトレースコマンドは、プロシジャごとに設定される標準スタックフレームに影響されることに注意してください。コンパイラの多くは、ローカル変数のないプロシジャを活用するためにスタックフレームを設定しません。その場合ス

タックトレースコマンドは、完全なトレース情報を表示しません（つまり、リンク情報が1個以上抜ける）。スタックトレースコマンドで完全な情報表示をするには、各プロシジャごとに標準スタックフレームを強制的に設定するためにコンパイラスイッチを使用する必要があります。

High Cコンパイラは、プロファイルファイル中か、各ソースファイルの先頭に文"pragma off(optimize_fp);" を置いてください。

K コマンドでは、各プロシジャに受け渡す引数の数は決まっていません。任意の数字（各プロシジャごとにリストする引数の数）をコマンドオペランドとして指定できます。オペランドがない場合、K コマンドは、プロシジャ引数を表示しません。プロシジャ引数は、386 ネイティブモードプログラムのデバッグ時、32ビットの値として表示され、リアルモードプログラムのデバッグ時では16ビットの値として表示されます。

KAコマンドは引数を伴い、各プロシジャの持つ引数の数を自動的に決定します。その結果、KAコマンドは、各プロシジャの正確な引数の数を表示できます。このコマンドは、表示する引数の数を自動的に決めるので、コマンドオペランドは必要ありません。KAコマンドは、プロシジャコールの直後に引数をスタックからポップオフする呼出し元のプロシジャに左右されることに注意してください。したがって、呼び出されたプロシジャはリターン前にその引数をポップしなければならないという規則に従う言語（Pascalなど）と一緒に、KAコマンドを使用することはできません。また、引数のポップを遅らせるコンパイラ使用は、KAコマンドの使用時には無効にしなければなりません。High Cコンパイラの場合、プロファイルファイル中か各ソースコードファイルの先頭に文“"pragma off (postpone_callee_pops);" を置いてください。

K コマンドとKAコマンドは、正しく機能するためには、デバッグ中のプログラムのメモリモデルが正しく指定されていなければなりません。メモリモデルは、-SMALL 、-COMPACT、-MEDIUM 、 -LARGE の各コマンドラインスイッチか、MS、MC、MM、MLのデバッガコマンドで指定できます。

例：

```
-K
get_line
scan_line+0B5
read_file+012C
main+03A

-K 2
get_line (000000001,0002AF44)
```

```
scan_line+0B5 (0000C252,FFFFFFFF)
read_file+012C (00000000,00000000)
main+03A (00000228,00000003)

-KA
get_line (000000001,0002AF44)
scan_line+0B5 (0000C252)
read_file+012C (00000000,00000000,0000A400,00000002)
main+03A (00000228,00000003)
```

3.11 M コマンド

\- Move Memory（メモリの移動）

構文：　M range address

説明：　M コマンドは、メモリブロックを別ロケーションにコピーするのに使用されます。このコマンドには三つのオペランドが必要です。最初の二つのオペランドは、コピーされるブロックのアドレス領域を指定します。3番目のオペランドは、そのブロックをコピーする位置を指定します。ソースブロックと移動先ブロックは、オーバラップできません。オーバラップしている場合には、移動先ブロックの内容は予測できないものとなります。開始アドレスでも、移動先アドレスでセグメントセレクタが指定されない場合、DSの示すセグメントが、デフォルトとして想定されます。

例：　-M 100 1FF 200

DS:100から1FF までのロケーションをDS:200から2FF までに移動する。

-M buf1　buf1+20　buf2

"buf1"の先頭32バイトを"buf2"の最初にコピーする。

3.12 MC、ML、MM、MS、M?コマンド

\- Select Memory Model（メモリモデルの選択）

構文：　MC

ML

MM

MS

M?

説明：　メモリモデル選択コマンドは、デバッグ中のプログラムのメモリモデルを指定します。K コマンド、KAコマンド（スタックトレース）およびポインタ間接式演算子 "*"を正しく使用させるために、メモリモデルの指定が必要です。
MSコマンドは、スモールメモリモデル（NEARコードポインタとNEARデータポインタ）を選択します。MCコマンドは、コンパクトメモリモデル（NEARコードポインタとFAR データポインタ）を選択します。MMコマンドは、ミディアムメモリモデル（FAR コードポインタとNEARデータポインタ）を選択します。また、MLコマンドは、ラージメモリモデル（FAR コードポインタとFAR データポインタ）を選択します。デフォルトでは、 386 | DEBUG は、スモールメモリモデルの指定があったものと想定します。

M? コマンドは、現在想定されているメモリモデルを表示するのに使用します。

例：　-MM　中型メモリモデルを指定する。
-ML　大型メモリモデルを指定する。
-M?
Current memory model=LARGE

3.13 O コマンド

- Output to Port（ポートへの出力）

構文：　O Port values

説明：　O コマンドは、データバイトを出力ポートに出力するのに使用されます。このコマンドは二つのオペランドをとります。最初のオペランドはデータバイトを送るポート番号、２番目のオペランドはポートに書き込むデータバイトです。
注意）このコマンドを誤って使用するとシステムが破壊され、リブートが必要になります。使用時には、必ず、そのポート番号とデータバイトが正しいかどうかを確認してください。

例：　-O 100 FF
バイト値FFをポート100 に出力する。

3.14 P 、PQコマンド

- Procedure Trace（CALL命令を越えるシングルステップトレース）

構文：　P [=address] [count]
PQ [=address] [count]

説明：　プロシジャトレースコマンドは、プログラムの実行を１度に１命令ずつCALL命令を越えてトレースするのに使われます。ただしCALL命令で呼び出されたプロシジャへは進みません。このコマンドは、トレースしているプロシジャの実行内容のみが知りたくて、それに呼び出されるプロシジャには興味がない場合に役に立ちます。

P コマンドは、T コマンドと同じように動作します。T コマンドと異なる点は、CALL命令を越えてトレースし、呼び出されたプロシジャへは進まないことです。同様に、PQコマンドは、TQコマンドと同じように動作します。TQコマンドと異なる点は、PQコマンドはCALL命令を越えてトレースすることです。T コマンドとTQコマンドについては3.20節を参照してください。

例：　-PT 3

CS:EIPから三つの命令をトレースする。

-PQA

10個の命令をトレースし、最後にレジスタの内容を表示する。

3.15 PD、PTコマンド

- Display Page Tables（ページテーブルの表示）

構文：　PD [range]

PT table [range]

説明：　PDコマンドは、システムページディレクトリを表示するのに使用されます。PTコマンドは、システムページテーブルの一つを表示するのに使用されます。この二つのコマンドは、表示するテーブルエントリの開始インデックスと終了インデックスの二つのオペランド（オプション）を持ちます。インデックス番号０は、テーブルの最初のエントリを、１は２番目の欄をというように指定します。

オペランドの指定がない場合、16個のテーブルエントリが表示されます。このエントリの表示は、前回の本コマンド使用時、最後に表示されたエントリの次から始まります。開始インデックスだけ指定された場合は、指定インデックスのエントリから16個のエントリが表示されます。開始インデックスと、終了インデックスの両方が指定された場合、開始インデックスから終了インデックスまでのすべてのエントリが表示されます。

PTコマンドは、必須のオペランドとして、表示するページテーブルのページディレクトリ中のインデックス番号をとります。ページディレクトリとページテーブルは、80386 がリニアアドレスを物理アドレスへ変換するのに使用されます。詳細は、80386 プログラマ用文法書を参照してください。ページディレクトリの各エントリにはページテーブルがマッピングされ、ページテーブルの各エントリには物理メモリページがマッピングされます。PDコマンドとPTコマンドは、テーブルエントリごとに以下の情報を表示します。値はすべて16進数で表示されるます。

・エントリのインデックス番号
・マッピングされたメモリページの物理ベースアドレス
・メモリページに対して許されたアクセス(“read”:読出し専用、“rd/wr”:読取り書込みアクセス)
・ページの使用状況(“none”:ページにアクセスされていない、“read”:ページは読取られたが書込まれていない、“write”:ページは書込まれた) 80386 は、ページテーブルの読取りアクセスと、書込みアクセスとの区別をしないので、ページディレクトリのエントリの使用状況は、“none” か “rd/wr”です
・ページアクセスに必要な特権レベル(“user”:すべての特権レベルがアクセス可能な場合、“supervisor”:プログラムは特権レベル 0.1または2で実行中でなければならない)

PDコマンドとPTコマンドは、リアルモードのデバッグ時は使用できません。

例：

```
-PD 0 1
#0000 Base addr=00045000 access=rd/wr usage=rd/wr privilege=user
#0001 Not present-value=00000000
-PT 0 15 15
#0015 Base addr=0005A000 access=rd/wr usage=none privilege=user
```

3.16 Q コマンド

\- Quit (終了)

構文： Q

説明： Q コマンドは、デバッガを終了するのに使用されます。他のプログラムを実行するには、このコマンドを使用してデバッグセッションを終了させて、DOS に戻します。

例：

```
-Q
C >(DOSプロンプト)
```

3.17 R コマンド

- Display or Change Registers（レジスタの表示または変更）

構文：　R [register] [value]

説明：　R コマンドは、内部80386 レジスタの内容を表示または変更するのに使用されます。このコマンドには三つの形があります。

まず、オペランドを指定しない場合、すべての80386 汎用レジスタが画面に表示されます。レジスタ内容は次のように表示されます。

```
-R
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00001FFE
DS=0014 SS=0014 ES=0014 FS=0014 GS=0014
CS:EIP=000C:0000200 EFLAGS=00000202 NV UP EI PL NZ NA PO NC
000C:0000200 B409 MOV AH,09
```

1行目は4個のデータレジスタEAX 、EBX 、ECX 、EDXを、2行目は4個のアドレスレジスタESI 、EDI 、EBP 、ESPを示しています。3行目は6個のセグメントレジスタの内の5個のレジスタDS、SS、ES、FS、GSを表示しています。4行目には、次に実行する命令のアドレス(CS:EIP)とフラグレジスタEFLAGSの内容が表示されています。EFLAGSは、16進値と最も重要なフラグビットのニーモニックの両方で表示されます。以下のニーモニックは、フラグビットがオンかオフかを示すために使用されます。

ビット番号	フラグ	オフ	オン
17	仮想8086モード	空白	VM
11	オーバーフロー	NV	OV
10	方向	UP	DN
9	割込み有効	DI	EI
7	符号	PL	NG
6	ゼロ	NZ	ZR
4	補助けた上げ	NA	AC
2	パリティ	PO	PE
0	けた上げ	NC	CY

2番目の形式のR コマンドは、単一の80386 レジスタの内容表示に使用されます。この形ではコマンドオペランドは表示するレジスタ名です。有効なレスジタ名を以下に示します。

32ビットデータレジスタ	EAX 、EBX 、ECX 、EDX
16ビットデータレジスタ	AX 、X 、CX、 DX
8 ビットデータレジスタ	AL、 BL 、CL、 DL、AH、 BH 、CH、DH
32ビットアドレスレジスタ	ESI 、EDI 、EBP 、ESP
16ビットアドレスレジスタ	SI、 DI 、BP、 SP
セグメントレジスタ	CS、 DS 、SS、 ES 、FS、 GS
命令ポインタ	EIP 、IP
制御レジスタ	CR0 、CR2 、CR3
デバッグレジスタ	DR0 、DR1 、DR2 、DR3 、DR6 、DR7
テストレジスタ	TR6 、TR7

3 番目の形式のR コマンドは、80386 レジスタの内容を変更するために使用されます。この形のR コマンドは、オペランドが二つ必要です。最初のオペランドは変更するレジスタ名であり、2 番目のオペランドはレジスタにロードする値です。

なお、386 ネィティブモードプログラムの実行中にCTRL/Cを入力してデバッガに制御を戻した場合、 386|DEBUG は、レジスタ値を変更できないようにすることに注意してください。ブレイクポイントとウォッチポイントは、CTRL/C入力後も通常通り設定できます。

例：

```
-R ESI
ESI=00000000
-R ESI 123
-R ESI
ESI=00000123
```

3.18 RXコマンド

- Display Extended Registers （拡張レジスタの表示）

構文： RX

説明： RXコマンドは、80386CPUのすべての内部レジスタを表示するのに使用されます。RXコマンドは、R コマンドと同様に、汎用レジスタ（EAX 、EBX など）、セグメントレジスタ(DS 、CSなど）およびフラグを表示します。さらに、RXコマンドは以下に示す80386 専用レジスタも表示します。

・デバッグレジスタ DR0、 DR1、 DR2、 DR3、 DR6、 DR7
・制御レジスタ CR0、 CR2、 CR3
・テストレジスタ TR6、 TR7
・GDTR、LDTR、IDTRの各レジスタ

例：

```
-RX
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00001FFE
DS=0014 SS=0014 ES=0014 FS=0014 GS=0014
CS:EIP=000C:0000200  NV UP EI PL NZ NA PO
NC
CR0=00000001 CR2=00000000 CR3=0000F000
DR0=00021360 DR1=00000000 DR2=00000000 DR3=00000000
DR6=0000E00F DR7=00000003
TR6=00000000 TR7=00000000
GDTbase=00011D96 GDTlimit=002F IDTbase=00011EF4
IDTlimit=07FF
LDT=0028 TR=0000
```

3.19 S コマンド

- Search Memory for String （メモリ中の文字列探索）

構文： S range values

説明： Sコマンドは、メモリブロック内のデータバイトの列を探索するのに使用されます。最初の二つのオペランドは、検索するメモリブロックのアドレス領域を指定します。残りのオペランドは、探索するメモリのバイト列を指定します。値は、単独のバイトまたは文字列として与えることができます。開始アドレスでセグメントセレクタを指定しなかった場合、DSの示すセグメントを指定したものと想定されます。

S コマンドは、指定されたバイトまたはバイト列を求めて指定された全領域を検索します。指定値が見つかるたびに、値列の最初のバイトアドレスが表示されます。

例：

```
-DB 0 1F
0014:00000000 B4 09 BA 10 00 00 00 CD-21 66 B8 00 4C CD 21 00
0014:00000010 48 65 6C 6C 6F 20 77 6F-72 6C 64 21 21 21 21 21 Hello worl
d!!!!!
-S 0 1F CD 21
0014:00000007
0014:0000000D
-S DS:0 1F 'HELLO'
-S DS:0 1F 'Hello'
0014:00000010
```

```
-S 14:0 1F'!'
0014:00000008
0014:0000000E
0014:0000001B
0014:0000001C
0014:0000001D
0014:0000001E
0014:0000001F
```

3.20 T 、TQコマンド

- Trace(Single-Step)(トレース（単一ステップ））

構文：　T [=address] [count]
　　　　TQ [=address] [count]

説明：　トレースコマンドは、プログラムの実行を1命令ずつトレースするのに使用されます。最も単純な形では、T コマンドはオペランドをとりません。この場合は、80386 命令を一つ実行し、その次に80386 レジスタの内容を画面に表示します。そのレジスタ内容はR コマンドと同じ形式で表示されます。また、命令ポインタGS:EIPによってアドレス指示された命令は実行されます。

T コマンドのオペランドとしてcount 数をとると、指定された数だけ命令をトレースします。この場合も、80386 レジスタの内容は各命令の実行後に表示されます。この形のT コマンドは、何度もT コマンドを入力することなしにプログラムをトレースするのに便利です。"*"文字は、無限値を指定するのに使用されます。無限値の指定がある場合、プロセッサ例外が発生するかプログラムが終了するまで、命令はトレースされます。

TQ(trace quietly) コマンドは、T コマンドとほぼ同じ動作をします。異なる点は、T コマンドでは各命令のトレースごとにレジスタ内容が表示されますが、TQコマンドでは、指定された数の命令のトレース後か、プロセッサ例外の発生後にだけレジスタ内容が表示されることです。デバッガは、TQ命令のあとに制御を獲得すると、トレースされた命令の数を表示します。

T コマンドとTQコマンドの2番目のオペランドとして、実行開始アドレスを指定することができます。実行開始アドレスの前に等号（=）を入力して指定します。開始アドレスに対してセグメントセレクタが指定されてない場合、CSレジスタのセレクタがデフォルトとして指定されます。

トレースされる命令がソフトウェア割込み(INT) 命令の場合、80386 アーキテクチュアでは、一つの命令ではなく二つの命令が実際にはトレースされます。T コマンドは、INT 命令のトレース後、メッセージ“stepped two instructions to...”を表示します。

例：　-T 5

CS:EIPの５個の命令をトレースする。

-TQ =100 A

CS:100の10個の命令をトレースし、10番目の命令のトレース後にだけレジスタ内を表示する。

-TQ *

エラーが発生するまで命令をトレースする。

3.21　U、U 16、U 32コマンド

\- Unassembly （逆アセンブリ）

構文：

```
U    [range]
U16  [range]
U32  [range]
```

説明：　U 、U16 、U32 の各コマンドは、メモリブロックを逆アセンブルし、その内容を80386 命令として表示するのに使用されます。これらのコマンドは、逆アセンブルするメモリ領域を指定する二つのオペランド（オプション）をとります。オペランドの指定がない場合、32バイトが逆アセンブルされます。逆アセンブルは、前回このコマンドが実行された時、逆アセンブルされた領域の最後のバイトの次から始まります。開始アドレスだけが指定された場合、開始アドレスから32バイトが逆アセンブルされます。開始アドレスと終了アドレスの両方が指定された場合、そのブロック全体が逆アセンブルされます。逆アセンブルの開始アドレスに対してセグメントセレクタが指定されていない場合、CSレジスタのセレクタがデフォルトとして使用されます。

386 | DEBUG は、逆アセンブルされた命令ごとに、命令アドレス、OPコードバイトの16進値、命令とオペランドに対するアセンブリ語文を１行に表示します。オペランドサイズは、必要に応じてPTR 演算子で指定されます。また、セグメントオーバライドは演算子“：”で表示されます。

開始アドレスの命令を逆アセンブルする前に、それ以前のシンボル中で最も近いシンボルからの命令のオフセットが表示される。unassembleコマンドが、一

群の命令の逆アセンブル中に新しい関数に入り込んだ場合、新しい関数の名前が表示される。

一群のコードを正しく逆アセンブルするために、 386|DEBUG は、各命令の実行時、コードセグメントディスクリプタ内のDビットがどの値を持つかを知る必要があります。Dビットは80386CPUが、命令オペランドとメモリアドレスのデフォルトサイズを決定するのに使用されます。Dビットが設定されている場合、80386 は、すべての変位とすべてのフルサイズオペランドがともに32ビット幅であると想定します。Dビットがクリアされている場合、16ビットサイズが想定されます。この値は80386 の命令OPコード解釈方法に影響するので、 386|DEBUG は同一の想定を行うために、Dビットの値を知る必要があるのです。

U16 形かU32 形の逆アセンブルコマンドが使用されている場合、 386|DEBUG は、コマンド名に基づいて、デフォルトのオペランドとアドレスのサイズを想定します。U 16は、 386|DEBUG に対しデフォルトのオペランド幅とアドレス幅を16ビットで想定するよう命じます。U 32の場合は省略時の幅を32ビットに想定します。

U 形の逆アセンブルコマンドは、デフォルトのオペランドサイズまたはアドレスサイズを指定しません。386|DEBUG は、次の論理を使って推論します。

- CPU がリアルモードの場合、16ビットのデフォルトサイズが想定されます。これは、 80386アーキテクチアの要件です。
- CPU が 386ネィティブモードの場合、 386|DEBUG は、逆アセンブル中のアドレスのセグメントセレクタを使ってそのサイズを想定します。 386|DEBUG は、ディスクリプタテーブル内のセレクタを検索し、Dビットをテーブルエントリから直接取り出します。

多くの場合、U 形の逆アセンブルコマンドで十分間に合います。U16 コマンドとU32 コマンドは、システムコードを書く時かデータセグメント内のコードの逆アセンブル時（たとえば、物理メモリの下位1メガバイトをマッピングするUSE32 データセグメント内のDOS コードを逆アセンブルするとき）に時々必要となります。

例： リアルモードの逆アセンブル例

```
-u MAIN MAIN+B
MAIN:
4E46:00000000 B84C4E        MOV    AX,4E4C
4E46:00000003 8ED8          MOV    DS,AX
4E46:00000005 66BA1F000000  MOV    EDX,0000001F
4E46:0000000B E81900        CALL   0027
```

386 ネィティブモードの逆アセンブル例

```
-u 2e40 2e48
SCAN+056:
000C:00002E40 0400          ADD   AL,00
000C:00002E42 00E8          ADD   AL,CH
000C:00002E44 A2070000B4    MOV   [B400007],AL
```

32ビットの逆アセンブルを強制的に実行する例

```
-U32 SCAN+6f SCAN+79
SCAN to 6F:
000C:00002A59 6650          PUSH  AX
000C:00002A5B 8D1DE9000000  LEA   EBX,[000000E9]
000C:00002A61 E8840B0000    CALL  000035EA
```

32ビットの386 ネィティブモードにしたままで16ビットコードを逆アセンブルする例（このコードはROM BIOSの一部である）

```
-U16 34:f000b f0010
0034:000F000B 8BF0          MOV   SI,AX
0034:000F000D B84000        MOV   AX,0040
0034:000F0010 8ED8          MOV   DS,AX
```

3.22 WC、WD、 WE 、WL、 WP コマンド
- Watchpoint Control （ウォッチポイント制御）

構文：

```
WC watchpts
WC*
WD watchpts
WD*
WE watchpts
WE*
WL
WP address datalen [r/w]
```

説明： ウォッチポイント制御コマンドは、データウォッチポイントの設定、クリア、有効化、無効化およびリスト作成を行うのに使用されます。データウォッチポイントは、特定メモリロケーションの読取り命令、書込み命令の実行直後アプリケーション実行を停止させます。データウォッチポイントが発生すると、デバッガは、アクセスされたメモリロケーションと現レジスタの内容とを表示したあとコマンドの入力を促します。データウォッチポイントは、プログラム中のコードまたはデータアドレスを破壊するようなバグを見つけるのに役に立ちます。

WPコマンドは、ウォッチポイントを設定するのに使用されます。最初のオペランドは、ウォッチポイントする位置アドレスで、２番目のオペランドは、アクセスの予想されるデータのサイズ（１バイト、２バイトまたは４バイト）です。誤ったデータサイズが使用された場合、ハードウェアは、そのメモリロケーションへのアクセスをしません。３番目のオペランドは、オプションであり、ウォッチポイントが書込みアクセス時に限って発生するか、どの種類のアクセス（読取り、書込みの両方）に対しても発生するかを指定します。英小文字“w”は、書込み専用ウォッチポイントを指定するのに使用されます。英小文字“r”は、読出し書込みウォッチポイントを指定するのに使用されます。３番目のオペランドの指定がない場合、書込み専用ウォッチポイントが設定されます。

WLコマンドは、WPコマンドで設定されたウォッチポイントをリストします。各ウォッチポイントは、 386｜DEBUG によって割り当てられる番号によって識別されます。WLコマンドは、ウォッチポイントごとに、(1)ウォッチポイント番号、(2)設定されたロケーションのアドレス、および(3)ウォッチポイントが書込み専用か読出し書込み用か、をリストします。さらに、ウォッチポイントが無効になっている場合、WLコマンドは、ウォッチポイント情報のあとに... disabled ... を表示します。

WDコマンドはウォッチポイントを無効にするのに使用され、WEコマンドは、ウォッチポイントを有効にするのに使用されます。両コマンドとも、ウォッチポイント番号のリストか、特殊記号“＊”（すべてのウォッチポイントを意味する記号）のいずれかを引数とします。WDコマンドは、活動して欲しくないウォッチポイントを一時的に無効にするのに使用され、WEコマンドは、WDコマンドですでに無効にされたウォッチポイントを有効にするために使用されます。

WCコマンドは、ウォッチポイントをクリアするのに使用されます。このコマンドは、ウォッチポイント番号のリストか、特殊記号“＊”(すべてのウォッチポイントを意味する記号）のいずれかを引数とします。ウォッチポイントは、いったんクリアされると、WLコマンドによってリストされることはなく、WEコマンドを使って有効にすることもできません。クリア後再度ウォッチポイントが必要になった場合、WPコマンドでそのウォッチポイントを再設定しなければなりません。

例：

```
-WP buf+21 1
-WP bufp 4 r
-WP errf 2 w
-WL
0 BUF+021 datalen=1 write only
1 BUFP datalen=4 read/write
```

```
2 ERRF datalen=2 write only
-WD 1
-WL
0 BUF+021 datalen=1 write only
1 BUFP datalen=4 read/write *** disabled***
2 ERRF datalen=2 write only
-WE *
-WC 0 2
-WL
1 BUFP datalen=4 read/write
```

3.23 X コマンド

- Display Symbol Table （シンボルテーブルの表示）

構文： X [symbol]

説明： X コマンドは、デバッグ中のプログラムのパブリックシンボルとそのシンボル値を表示するのに使用されます。X コマンドでコマンドオペランドが指定されない場合、シンボルテーブル全体が表示されます。文字列値がコマンドオペランドとして指定されている場合、指定文字列で始まる名前の全シンボルが表示されます。

例：
```
-X ma
MAIN = 0000000A
MAKE_PTR = 000032C
MARK_POS = 000045B0
- mak
MAKE_PTR = 0000032C
```

3.24 XSコマンド

- Display Segment Names（セグメント名の表示）

構文： XS

説明： セグメント名表示コマンドは、プログラムの各セグメントの名前、セグメントセレクタ値（セグメントをアドレス指定するためにセグメントレジスタにロードされる値）およびサイズ（単位：バイト）を表示します。386 ネイティブモードプログラムに対しては、リンカは、“PROGRAM”という名の単一のセグメントを作成し、セグメントセレクタ000Cと0014が、常にそのセグメントを指すために設定されます。したがって、XSコマンドは、 386ネイティブモードプログラムに対して

は、いつも同じ情報を表示します。リアルモードプログラムに対しては、ソースコードで定義されたのと同じようにセグメントとグループ名が表示されます。セグメント値は、そのセグメントかグループのパラグラフベースアドレスです。

例： 386 ネイティブモードプログラムの場合：

```
-XS
PROGRAM      000C Size = 0001A800
```

リアルモードプログラムの場合

```
-XS
_BSS         2E76 Size = 00000200
_CODE        2350 Size = 0000B260
_DATA        2E96 Size = 000039E4
DGROUP       2E76 Size = 00003BE4
```

3.25 XWコマンド

- Locate Symbol（シンボルの位置を知る）

構文： XW address

説明： XWコマンドは、プログラム中の特定アドレスより前にあるシンボルで最も近いシンボルを見つけるために使用されます。コマンドオペランドは、シンボルテーブルの中で検索する、プログラムアドレスです。指定アドレスより前で、最も近いシンボルと、そのシンボルのアドレスから指定アドレスまでのオフセットとが表示されます。指定されたアドレスがシンボルのアドレスと全く等しい場合、シンボルはオフセットなしで表示されます。

例：

```
-XW 400
CHK_ERR+01C
-XW dgroup:30
DATBUF
```

3.26 ? コマンド

- On Line Help

構文： ?

説明： ?コマンドは、オンラインヘルプを 386 | DEBUG コマンド形式で得るのに使用されます。ヘルプ画面には、各コマンドのリスト、各コマンドのオペランド、1行のコマンド説明が表示されます。

付録A 386|DEBUG コマンドライン

386|DEBUG switches name param(s)

switches	デバッガのデフォルト時動作をオーバライドするためのオプションコマンドラインスイッチ
name	デバッグする.EXPファイル名.EXEファイル名または.REXファイル名
param(s)	追加のコマンドラインパラメタをデバッグ中のプログラムに受け渡す

付録 B 386 | DEBUG コマンドの要約

386 | DEBUG コマンド:

BC numberlist |*	Clear breakpoints
BD numberlist |*	Disable breakpoints
BE numberlist |*	Enable breakpoints
BL	List all breakpoints
BP address	Set breakpoints
C range address	Compare memory
D [range]	Dump memory
DA [range]	Dump memory in ASCII
DB [range]	Dump memory as hex bytes
DD [range]	Dump memory as hex doublewords
DW [range]	Dump memory as hex words
DG [range]	Dump GDT
DI [range]	Dump IDT
DL [range]	Dump LDT
E address value(s)	Enter(change)memory by bytes
EB address value(s)	Enter(change)memory by bytes
ED address value(s)	Enter(change)memory by doublewords
EW address value(s)	Enter(change)memory by words
F range value(s)	Fill memory
G [=address] [address(es)]	GO (and set breakpoints)
H value value	Hex arithmetic
I port	Inpout from port
K [arg_count]	Display procedure call stack
KA	Display call stack with arguments
M range address	Move memory

M?	Show current memory model
MC	Assume compact memory model
ML	Assume large memory model
MM	Assume medium memory model
MS	Assume small memory model
O port value	Output to port
P	Trace across CALL instruction
PD [range]	Dump page directory
PQ	Trace quietly across CALL instructions
PT table [range]	Dump page table
Q	Quit
R [register] [value]	Display or change registers
RX	Dispaly extended registers
S range value(s)	Search memory for string
T [=address] [count]	Trace(single-step)
TQ [=address] [count]	Trace quietly
U [range]	Unassemble memory
U16 [range]	Unassemble memory in USE16 segment
U32 [range]	Unassemble memory in USE32 segment
WC numberlist | *	Clear watchpoints
WD numberlist | *	Disabel watchpoints
WE numberlist | *	Enable watchpoints
WL	List all watchpoints
WP addr datalen [r | w]	Set watchpoints
X [symbol]	Display symbol table
XS	Display segment names
XW address	Locate symbol at address

式形式：

値	12A	16進
	'ABC' AL F5+AA FF-12	文字列 レジスタの内容 加算 減算
アドレス	12A main CS:145 8:145 ESP SS:ESP but+10 EBP-10	オフセット シンボル セグメントレジスタオーバライド セレクタオーバライド レジスタを介した間接アドレス指定 セグメントオーバライドのあるレジスタを介した間接アドレス指定 加算 減算
領域	buf　buf+FF CS:100 1FF 14:100 1FF	単純領域 セグメントオーバライドのある領域 セレクタオーバライドのある領域

付録C エラーメッセージ

この付録では、デバッガに発生する可能性のあるエラー、その原因と修正方法について述べます。エラーメッセージは、二つに分けて説明します。C. 1では、 386｜DEBUG が最初のコマンドプロンプトの表示前にプログラムをロードしようとするとき発生するエラーを示します。C. 2では、コマンド入力時に発生するエラーを示します。エラーメッセージはアルファベット順になっています。

C.1 プログラムロードエラー

エラー： ? Can't open executable file:filename

原因： 存在しない.EXPファイル、.EXEファイルまたは.REXファイルをデバッグしようとした。

処置：
1）.EXPファイル名のスペルを訂正する。
2）.EXPファイルのあるディレクトリ名を正しく入力する。
3）386｜LINKを使用して、.EXPファイルを作成する。

エラー： ? Error reading file:filename

原因： .EXPファイルが損傷している。

処置： リンクエラーがないことを確認して、.EXPファイルを再構築する。

エラー： ? Insufficient memory to load program

原因： プログラムのサイズが使用可能なメモリより大きい。

処置：
1）プログラムを縮小する。
2）より大きなメモリを購入する。
3）メモリ常駐プログラムをシステムから除去する。

エラー：　? Load error
原因：　MS-DOSがリアルモード.EXEファイルをロードしようとしたときに、原因不明のエラーが発生した。
処置：　システムをリブートする。

エラー：　? Missing file name
原因：　デバッグするプログラムファイルを指定しないで 386 | DEBUG を実行しようとした。
処置：　デバッグするプログラムのある.EXPファイル名を与える。

エラー：　? Not an executable file:filename
原因：　ファイルが.EXP形式、.EXE形式または.REX形式のファイルではない。

C.2 コマンドエラー

エラー：　? Bad command operand
原因：　コマンドに対するオペランドの形式が間違っている。たとえば、

1G	(違法な16進数字)
1+	("+" の右側にオペランドがない)
'ABC	(右側の引用符がない)

処置：　オペランドを修正し、コマンドを再入力する。

エラー：　? Can't alter registers after CTRL/C in protected mode
原因：　デバッグ中のプログラムが386 ネイティブモードで実行中、CTRL/Cで制御獲得後80386 レジスタの内容を変更しようとした。この変更は禁止されている。

エラー：　? Extra characters at end of line
原因：　コマンド行の最後のコマンドオペランドのあとにまだ文字がある。
処置：　余分な文字を行から削除して、コマンドを再入力する。

エラー:　? Invalid address: segment:offset

原因：　無効なセグメントセレクタ値か、セグメント制限値を越えたオフセットのいずれかを持つアドレスが指定された。

エラー:　? Missing required command operand

原因：　コマンドに必要なオペランドが一つも指定されていない。

処置：　help(?) コマンドを使って、必要なオペランドを指定する。

エラー:　? Only count bytes could actually be entered

原因：　途中でセグメントの終りに達したため、要求より少ないバイト数が、いずれかの入力コマンド(E、EB、EW、ED)によって実際にメモリに書き込まれた。

エラー:　? Only count bytes were actually filled

原因：　セグメントの終りに達したため、要求より少ないバイト数が、fill(F) コマンドによってメモリに実際に書き込まれた。

エラー:　? Segment selecter: value not readable

原因：　指定されたセグメントの読取りができない。

処置：　同一のメモリをマッピングする別セグメントを使ってデータを読取る。

エラー：　? Segment selector: value not writable

原因：　指定されたセグメントの書き込みができない。

処置：　同一メモリーをマッピングする別セグメントを使ってデータを書き込む。

エラー：　? Unknown command

原因：　正しくないコマンド名を386 | DEBUG に与えた。

処置：　コマンド名のスペルを修正する。正しいコマンドの一覧表については付録Aを参照のこと。

エラー:　? Value out of range

原因：　コマンドオペランドとして指定された値が、コマンドの最大許容値をこえている。

索引

386 | ASM TOOL KIT　ユーザーズマニュアル
第3部　シンボリックデバッガ

英数・記号

あ行

か行

さ行

た行

な行

は行

ま行

ら行

【386|DEBUGコマンド】

第4部

ライブラリアン

386 | LIB

目次

386 | ASM TOOL KIT　ユーザーズマニュアル
第４部　ライブラリアン 386 | LIB

第1章 概要

386｜LIB は、OMF-86とEasy OMF-386のオブジェクト・ファイル用ライブラリアンです。

これはPhar Lap Software 社の『80386 ソフトウェア開発シリーズ』の一つで、Phar Lap Software 社のリンカ:386 LINK および LinkLocで処理できるオブジェクトライブラリファイルを、作成、修正、テストするために使用するものです。ここでは 386｜LIB の使用法を説明します。

第2章 386|LIBの使用法

2.1 コマンドライン構文法

ライブラリアンをするために使用するコマンドラインは、ライブラリアンタスクイメージ名の386LIBと、そのあとに続くファイル名のリストとスイッチから構成されます。スイッチはライブラリアンのデフォルトオプションを無効にします。ライブラリの処理中にエラーが発生するとスクリーン上にエラーメッセージが表示され、マップファイルがあればそれに書き込まれます。

ファイル名は、完全なファイル名（ファイル名.拡張子）か、386|LIB がデフォルト拡張子を提供している場合は拡張子なしで指定することができます。

何も指定しない場合、ライブラリアンは以下のようなファイル名拡張子を想定します。

ファイルタイプ	デフォルト拡張子
オブジェクトファイル	.OBJ
ライブラリファイル	.LIB
ライブラリバックアップファイル	.BAK
ライブラリマップファイル	.MAP
ライブラリアンコマンド・ファイル	.LBC

さらに、完全または部分的なパスを、ファイルと共に指定することができます。

何も指定しない場合は、現在のデフォルトデバイス、ディレクトリと見なされます。

スイッチは“-”(マイナス記号) ではじまり，そのあとにスイッチ名がきます。マイナス記号とスイッチ名の間にスペースを入れることはできません。引数はスペースを区切りとしてスイッチのすぐあとにきます。入力ファイル名やモジュール名は、スペースやコンマで区切って指定することができます。スイッチが続くとき、もしくはスイッチと入力ファイル名またはモジュール名が続くときは、スペースで区切らなくてはなりません。入力ファイル名とスイッチはコマンドライン上にどの順序で置いてもかまいません。入力ファイル名とモジュール名は頭にマイナス記号をもってくることはできないので、386|LIB はファイル名とスイッチを区別することができます。「コマンドラインスイッチ」の項に、色々な 386|LIB の動作を示すためのコマンドラインの例を入れてあります。

2.2　間接コマンドファイル

Phar Lap社の他製品と同様に、間接コマンドファイル中で共通的に使用される 386|LIB コマンドラインパラメータを置くことができます。 386|LIB コマンドラインパラメータは、コマンドラインで入力されるのと同じ方法で間接ファイルに入ります。間接コマンドファイル中のコマンドラインパラメータは、必要に応じた行数を入れることができます。間接コマンドファイルを使用するためには、コマンドライン上のファイル名の前に"@" を付けます。 386|LIB が"@" で始まるコマンドラインパラメータを認識すると、そのファイルをオープンし、ただちにその中のコマンドを処理します。

このファイルに拡張子が指定されていない場合、拡張子は".LBC"とみなされます。例をあげると

```
386LIB @comfile -add newmod
```

は "COMFILE.LBC"という名のファイル中のコマンドラインパラメータを、"-add newmod" パラメータを処理する前に処理します。

2.3　コマンドラインスイッチ

コマンドラインスイッチはデフォルトの指定を変更します。デフォルトでは 386|LIB は以下の仕様になっています。

- 入力ライブラリと同じ名前の出力ライブラリを生成する。
- 全てのシンボル名を大文字/ 小文字の区別なく処理する。
- 16バイトのライブラリページを使用する。
- ライブラリマップやバックアップファイルを生成しない。

コマンドラインスイッチは"-"(マイナス記号) で始まり、そのあとにスイッチ名がきます。各スイッチ名には一般型と省略型があります。引数はスペースを区切りとしてスイッチのすぐあとにきます。コマンドライン上に矛盾する複数のスイッチが置かれた場合は、一番最後 (右端) のスイッチが優先されます。

2.3.1　オブジェクトモジュールスイッチ

オブジェクトモジュールスイッチは、ライブラリからオブジェクトモジュールを引き出したり、加えたり、消去します。ファイル中に含まれているオブジェクトモジュールに働くスイッチは、オブジェクトモジュールを含んでいるファイル名を引数として使用します。ライブラリ中に含まれているモジュールに働くスイッチは、ライブラリ中のモジュール名を引数として使用します。 2.5節では 386|LIB が準じているモジュールの名付け規則を説明しています。スイッチがその引数としてファイル名を使用するかモジュール名を使用するかについては、それぞれのスイッチの説明を参照してください。

注意: オブジェクトファイル名を引数として使用するスイッチは、オブジェクトライブラリも引数として受け付けます。オブジェクトライブラリをスイッチ引数として指定すると、

ライブラリ中の各モジュールが適切に処理されます。これは 2つのライブラリファイルの内容をマージするのに便利です。

−CREATE

-CREATE スイッチは、スイッチ引数で指定したモジュールを使用してライブラリを新たに作ります。そのライブラリファイルがすでに存在する場合、その内容は新しいライブラリが作成される前に消されてしまいます。

構文： -CREATE ファイル1 ....ファイルn

省略形: -C ファイル1 ....ファイルn

例:
```
386lib newlib -create mod1.obj mod2.o mod3
386lib newlib -c ¥obj¥mod1 ¥newobj¥mod2 ¥obj¥mod3
```

−ADD

-ADDスイッチは、現在あるオブジェクトライブラリに新たなオブジェクトモジュールを付け加えるのに使用します。モジュールは-ADDスイッチの引数で指定されます。付け加えようとするモジュールがすでにライブラリ中に存在する場合は、エラーメッセージが表示され更新は行われません。指定されたライブラリが存在しない場合は、ワーニングエラーが表示され指定したモジュールを付け加える前にそのライブラリを作成します。

構文： -ADD ファイル1.... ファイルn

省略形：-A ファイル1.... ファイルn

例：
```
386lib mlib -add newmod
386lib mlib -a ¥obj¥nm.obj ¥newobj¥mod2 ¥obj¥mod3
```

−DELETE

-DELETE スイッチは、処理中のライブラリから指定したモジュールを除去します。-DELETEスイッチは、引数としてファイル名ではなくモジュール名を使用します。引数はすでにライブラリ中に挿入されているモジュールを参照するからです。

構文： -DELETE モジュール1.... モジュールn

省略形：-D モジュール1.... モジュールn

例：
```
386lib mathlib -delete trigfunc
386lib mlib -d sin cos tan
```

－ＲＥＰＬＡＣＥ

-REPLACEスイッチは、ライブラリ中のオブジェクトモジュールを置き換えます。

スイッチのあとに、置き換えるオブジェクトモジュールを含んだファイル名がきます。置き換えるモジュールがすでにライブラリ中に存在する場合は、それを消去し新たなオブジェクトモジュールを付け加えます。ライブラリ中にモジュールが存在しない場合、ワーニングエラーを表示しそのモジュールをライブラリに付け加えます。指定されたライブラリが存在しない場合は、ワーニングエラーを表示し、指定したモジュールを付け加える前にライブラリを作成します。

構文：　-REPLACE　ファイル1.... ファイルn

省略形：-R　ファイル1.... ファイルn

例：

```
386lib clib -replace newmod
386lib hce -r ¥newobj¥mod2  ¥nobj3¥newmod3.o
```

－ＥＸＴＲＡＣＴ

-EXTRACTスイッチは、ライブラリからオブジェクトモジュールのコピーを引き出します。このスイッチのあとに、引き出されるモジュール名のリストが続きます。引き出される各モジュールは、そのモジュール名にデフォルトのオブジェクトファイル拡張子が付いたファイル名のファイルに置かれます。ライブラリからオブジェクトモジュールを引き出すことによって、元のライブラリが変更されることは全くありません。

構文：　-EXTRACT　モジュール1 ....モジュールn

省略形：-E　モジュール1 ....モジュールn

例：

```
386lib mathlib -extract trigfunc
386lib mlib -e sin cos tan
```

－ＥＸＴＲＡＣＴＡＬＬ

-EXTRACTALL スイッチは、ライブラリ中に含まれる全てのオブジェクトモジュールのコピーを引き出します。このスイッチは-EXTRACTスイッチでライブラリ中の各モジュールを指定したときと全く同じ働きをします。引き出される各モジュールは、デフォルトのオブジェクトファイル拡張子のついたモジュール名と同じ名前のファイルに置かれます。

-EXTRACTスイッチ同様に、-EXTRACTALL スイッチも元のライブラリを変更することは全くありません。

構文： -EXTRACTALL

省略形：-EALL

例：　386lib mathlib -extractall
　　　386lib mlib -eall

2.3.2　マップファイルスイッチ

－MAP

-MAPスイッチは、各モジュール中で定義されている全てのパブリックシンボルのリストと共に、ライブラリ中の全てのオブジェクトモジュールのリストを含んだライブラリマップファイルを作成します。スイッチのあとにはマップファイル名が続きます。指定されたマップファイル名が拡張子を含まない場合、デフォルトである".MAP"が拡張子となります。ライブラリマップファイルのフォーマットについては本書の2.6節で説明します。

構文： -MAP　ファイル名

省略形：-M　ファイル名

例：　386lib mathlib -map mathlib
　　　386lib libc -m libc.lmp

－NOMAP

-NOMAPスイッチは、処理中のライブラリに対するライブラリマップが生成されないようにします。これは386|LIB のデフォルトの指定でもあります。

構文： -NOMAP

省略形：-NOM

例：　386lib mathlib -extract sin -nomap
　　　386lib mlib -delete cos -nom

2.3.3　ライブラリバックアップファイルスイッチ

－ＢＡＣＫＵＰ

-BACKUPスイッチは、ライブラリに修正を加える前にそのライブラリのバックアップコピーを作成します。バックアップファイルの名前はライブラリ名に".BAK"拡張子を付けたものになります。

構文：　-BACKUP

省略形：-B

例：
```
386lib libc -replace printf -backup
386lib mlib -delete cos -b
```

－ＮＯＢＡＣＫＵＰ

-NOBACKUPスイッチはライブラリに修正を加える前にそのライブラリのバックアップコピーを作成しません。これは386 | LIB のデフォルトの指定でもあります。

構文：　-NOBACKUP

省略形：-NOB

例：
```
386lib libc -replace printf -nobackup
386lib mlib -delete cos -nob
```

2.3.4　ライブラリページサイズスイッチ

－ＰＡＧＥＳＩＺＥ

-PAGESIZE スイッチは、ライブラリファイルのライブラリページサイズを指定します。このスイッチを指定しないと、処理されるライブラリの現在のページサイズを使用します。ライブラリの作成中に-PAGESIZEを指定しない場合は、386 | LIB はライブラリページサイズのデフォルト値、16バイトを想定します。ページサイズは、ライブラリ中にオブジェクトモジュールが並んでおり、ライブラリサイズの最大値を限定しているときに使用されます。大きなページサイズを指定すれば、より大きなライブラリサイズの最大値を得られますが、その分ライブラリ中のオブジェクトモジュール間でスペースを消費します。-PAGESIZE スイッチパラメータは16から512 までの 2の巾乗数でなければなりません。ページサイズと許されるライブラリサイズの最大値は以下の表の通りです。

ページサイズ	ライブラリサイズの最大値
16	1 メガバイト
32	2 メガバイト
64	4 メガバイト
128	8 メガバイト
256	16 メガバイト
512	32 メガバイト

ライブラリページサイズパラメータは、ライブラリの修正を行う時にも使用されます。つまり、-PAGESIZE スイッチを使用することによって、ファイルを修正している最中に存在するライブラリファイルのページサイズを変えることが可能です。

構文： -PAGESIZE サイズ

省略形：-PAGESIZE サイズ

例： 64バイトのページサイズでライブラリを作成する場合

```
386lib newlib -create mod1.obj mod2.o mod3 -pagesize 64
```

"small.lib"というライブラリのページサイズを 128バイトに変更し、新しいライブラリに"large.lib" と名付ける場合

```
386lib large -create small -pagesize 128
```

2.3.5 大文字／小文字判別スイッチ

デフォルトでは386 | LIB はユーザ定義のシンボルに対して大文字／小文字を区別しません。たとえば、"my _ sym"も"my _ SYM"も"MY _ SYM"も 386 | LIB では全て同じと見なされます。以下の大文字／小文字を区別するスイッチを指定すると上記例の三つのシンボルは別のものと見なされます。

－TWOCASE

-TWOCASEスイッチはユーザ定義のシンボルを大文字／小文字の区別をして処理するようにします。このスイッチが使用されると同じアルファベット文字の大文字と小文字が区別されます。

構文： -TWOCASE

省略形：-TC

例：　386lib libc -replace printf -twocase
　　　386lib mlib -add cos -tc

－ONECASE

-ONECASEスイッチはユーザ定義のシンボルの大文字／小文字の区別をしないようにします。これは 386|LIB のデフォルトの指定と同じことです。

構文：　-ONECASE

省略形：-OC

例：　386lib slibc -replace scanf -onecase
　　　386lib mlib -add cos -oc

2.4　ライブラリアン動作の順序

386|LIB がライブラリ中で複数の動作を行うとき、前もって決まった順序で実行されます。

ユーザは普段はこの順序に対して特に考慮する必要はありません。ライブラリアン動作の順序は以下のようになります。

1. 指定されたオブジェクトライブラリがまだ存在しない場合は、まずそれを作成します。
2. 存在するライブラリからモジュールを引き出す場合は、モジュールはライブラリが修正される前に引き出されます。
3. 消去されるか置き換えられるモジュールは、ライブラリから除去されます。
4. ライブラリの修正は、新たなモジュールをライブラリに付け加えることによって完了します。
5. -MAPスイッチが指定されていると、最後にライブラリマップファイルが生成されれます。

2.5　モジュール名設定規則

386|LIB がライブラリにオブジェクトモジュールを付け加えるときは、常にオブジェクトモジュール中のモジュール名レコードを、オブジェクトモジュールを含むファイルのベース名に設定します。この特徴はMicrosoft のライブラリアンと互換をとるためのもので、言語翻訳プログラムが正しいモジュール名情報を生成しない場合に役に立ちます。
たとえば

　　"c:¥ 386obj¥ startup.obj"

というDOS ファイルに含まれているモジュールは、オブジェクトモジュールの持っていた元の名前とは無関係に、"startup" と設定されたモジュール名レコードでライブラリに付加されます。

386|LIB は、同じライブラリフォーマットを持ちながら異なったモジュール名設定規則を使用するライブラリアンで作成されたオブジェクトライブラリも処理することができます。 386|LIB のモジュール名設定規則に従っていないオブジェクトモジュールに出会った場合は、モジュール名レコードからベースファイル名を引き出し、モジュール名を必要とするスイッチでその名前を使用できるようにします。したがって

"d:¥obj¥new¥recalc.asm"

というモジュール名レコードを持ったオブジェクトモジュールは、モジュール名を必要とするスイッチで"recalc"を使用することによって参照されます。

2.6 ライブラリマップ記述

ライブラリアンで作成されるマップファイルは、各モジュールで定義されたパブリックシンボルのリストと共に、ライブラリ中にオブジェクトモジュールのリストを含みます。

マップの先頭にあるヘッダが386|LIB のバージョンナンバと処理されるライブラリ名を識別します。各モジュールのパブリックシンボルリストは、モジュール中で定義されたシンボルと同様に並べられます。モジュール中での各ルーチンの位置の情報はリンク時に計算されるため、このマップ中には含まれません。次に例を示します。これは"init.obj"と"testmain.obj"という二つのオブジェクトモジュールから構成される"test.lib"という小さなライブラリのライブラリマップファイルです。

```
386 LIB:1.1z--Copyright (c)1986-88 Phar Lap Software,Inc.
Library File - test.lib

Module:   init
      _start
      _exit

Module:   testmain
      _main
      _subl
      _recalc
      _redisp
      _makecell
      _delcell
```

付録A コマンドラインスイッチ

コマンドライン・スイッチ		機　　能
386LIB	libname	ライブラリ名〔.ＬＩＢ〕
	-CREATE ファイル名…	オブジェクトファイルからライブラリを作成する
	-ADD ファイル名…	新たなオブジェクトモジュールをライブラリに付け加える
	-REPLACE ファイル名…	ライブラリ中のオブジェクトモジュールを置き換える
	-DELETE モジュール名…	オブジェクトモジュールをライブラリから消去する
	-EXTRACT モジュール名…	オブジェクトモジュールをライブラリから引き出す
	-EXTRACTALL	全てのオブジェクトモジュールをライブラリから引き出す
	-BACKUP	バックアップライブラリファイルを作る
	-NOBACKUP	バックアップライブラリファイルを作らない
	-MAP ファイル名	ライブラリマップファイル〔.MAP〕を作成する
	-NOMAP	ライブラリマップファイルを作成しない
	-PAGESIZE n	ライブラリのページサイズをn とする

コマンドライン・スイッチ	機　　能
-ONECASE	シンボル処理で大文字／小文字の判別をしない
-TWOCASE	シンボル処理で大文字／小文字の判別をする
@ファイル名	間接コマンドファイル［.LBC］をオープンする

付録B エラーメッセージ

ライブラリ処理中にエラーがおきた場合、386|LIB はエラーを確認し、その重要度とエラーメッセージをスクリーンに表示します。エラーがモジュールやユーザ定義のシンボルによる場合、そのモジュール名またはシンボル名が表示されます。ライブラリマップファイルが作成される場合は、同じエラーメッセージがマップファイル中にも記されます。

エラーは、ワーニングエラー、エラー、フォータルエラーの 3つに分類されます。ワーニングエラーは、ライブラリの保全が脅かされることはなく、無視することもできるエラーです。このエラーがおきても、他の処理は行われます。エラー、フォータルエラーは、正常でないライブラリが作成されたためにおこるエラーです。エラー、フォータルエラーがおこるとライブラリファイルは更新されません。

以下にエラーメッセージを上記の三種類に分けて説明します。

B.1 ワーニングエラー

Warning: Library file "XXX" does not exist so it will be created

原因： 存在しないライブラリファイルにオブジェクトモジュールを追加するように命令されたため。このエラーがおこると、386|LIB は自動的にライブラリを作成し、指定されたモジュールを加えます。

処置： ワーニングエラーなので、無視できます。-CREATE を使用してライブラリを作成することによってこのワーニングエラーを避けることができます。

Warning: Module "XXX" not found -- delete ignored

原因： ライブラリ中にないモジュールを消去しようとしたため。

処置： ライブラリ中のモジュール名の綴りをチェックしてください。

Warning: Module "XXX" not found -- extract ignored

原因： ライブラリ中にないモジュールを引き出そうとしたため。

処置： ライブラリ中のモジュール名の綴りをチェックしてください。

Warning: Module "XXX" not found -- module was added, not replaced

原因： ライブラリ中にないモジュールを置き換えようとしたため。

処置： ワーニングエラーなので、無視できます。ライブラリ中のモジュール名の綴りをチェックしてください。

B.2 エラー

Error: Because of errors, library file "XXX" has not been updated

原因： このエラーメッセージは、処理中のライブラリファイルが正常でないため更新されていないことを、他のエラーとともに表示します。

処置： 表示された他の全てのエラーを、修正してください。

Error: Cannot create object file for module "XXX" -- invalid file name.

原因： 386|LIB がライブラリから引き出されるモジュールに対して使用するファイル名が適切でないため。386|LIBはこのファイル名を、ライブラリ中のモジュール名にデフォルトオブジェクトファイルの拡張子を付けて作成します。

Error: Duplicate definition of symbol "XXX" in file "YYY"

原因： パブリックシンボルが複数のオブジェクトモジュール中で定義されている。

処置： 1)問題のパブリックシンボル名を変更すれば、他のパブリックシンボル名とぶつかりません。

2)モジュールからパブリックシンボルを消去します。

3)シンボルを定義しているオブジェクトモジュールの一つをライブラリ中に置かないようにします。

Error: Module "XXX" is already in the library

原因：　ライブラリ中にすでに存在しているモジュールと同じ名前のオブジェクトモジュールを、追加しようとしたため。

処置：　1)オブジェクトファイル名を変更します。

2)-ADDスイッチの代わりに-REPLACEスイッチを使って、新しいオブジェクトモジュールを追加する前に旧オブジェクトモジュールをライブラリから消去します。

B.3　フォータルエラー

Fatal Error: Bad library file "XXX"

原因：　指定したライブラリファイルが妥当なものではなかったため。

処置：　指定したファイルがPhar LapまたはMicrosoft のライブラリファイルのフォーマットと適合しているかどうか確認してください。

Fatal Error: Bad number specified on the command line -- "XXX"

原因：　コマンドスイッチ引数として不当な数字が指定されたため。このエラーは数字がまちがった型（不適切な数文字）であるか、スイッチの指定範囲を超えているためにおこるものです。

処置：　指定した数字が有効範囲かどうか確認してください。

FatalError: Bad object module in file "XXX"

原因：　386 | LIB の動作が不適切なオブジェクトモジュールに対して行われようとしたため。

処置：　1)オブジェクトモジュールがOMF-86またはEasy OMF-386ファイルに適したものかどうか確認してください。

2)ファイルのコンパイル中またはアセンブリング中にエラーがおこった場合は、オブジェクトファイルをライブラリに付加する前にエラーを修正してください。

3)Phar Lapのリンカ(386LINKかLinkLoc)でリンクすることによって、オブジェクトファイルの誤りについての詳細を得ることができます。リンカは"invalid object file"という同義のメッセージを表示し、続いてファイルが不適当な理由を表示してくれます。

Fatal Error: Cannot create "XXX" file -- その理由

原因： オペレーティングシステムエラーにより、386 LIB は"XXX" という名前のファイルを作成することができなかったため。

処置： オペレーションシステムのマニュアルを調べて、エラーの原因を解明し、それを取り除いてください。

Fatal Error: Cannot open "XXX" file --その理由

原因： オペレーティングシステムエラーにより、386 | LIBは"XXX" という名前のファイルをオープンすることができなかったため。

処置： オペレーションシステムのマニュアルを調べて、エラーの原因を解明し、それを取り除いてください。

Fatal Error: Command too long -- "XXX"

原因： 間接コマンドファイル中で指定したコマンドラインパラメータが 255文字より長いため。

処置： パラメータを短くしてください。

Fatal Error: Error reading from "XXX" file --その理由

原因： オペレーティングシステムエラーにより、386 | LIBは"XXX" という名前のファイルからデータを読み込むことができなかったため。

処置： オペレーションシステムのマニュアルを調べて、エラーの原因を解明し、それを取り除いてください。

Fatal Error: Error writing from "XXX" file --その理由

原因： オペレーティングシステムエラーにより、386 | LIB は"XXX" という名前のファイルにデータを書き込むことができなかったため。

処置： オペレーションシステムのマニュアルを調べて、エラーの原因を解明し、それを取り除いてください。

Fatal Error: Extra command line argument -- "XXX"

原因： コマンドライン上に複数のライブラリファイルを指定したか、余計なスイッチ引数を指定したため。

処置： コマンドラインパラメータをチェックし、無関係な引数を取り除いてください。

Fatal Error: Invalid file name -- "XXX"

原因：　コマンドライン上に、ファイル名の正しい構文法に従わないファイル名が指定されているため。

処置：　適切なファイル名を使用してください。

Fatal Error: Library file has overflowed -- increase page size

原因：　ライブラリファイルのサイズが、ライブラリページサイズの 65536倍を超えたため。

処置：　-PAGESIZE スイッチを使用して、より大きなページサイズを指定してください。一度ライブラリのページサイズを更新して使用できれば、再びオーバーフローが起きるまでは-PAGESIZE スイッチを指定する必要はありません。

Fatal Error: Missing value to command switch -- "XXX"

原因：　386 | LIB コマンドスイッチで引数が必要なものが引数なしで指定されているため。

処置：　スイッチに値を指定してください。

Fatal Error: One command file cannot call another command file -- "XXX"

原因：　間接コマンドファイルを他の間接コマンドファイル中でオープンしようとしたため。386 | LIBは間接コマンドファイルの入れ子をサポートしていません。

処置：　1)二つのコマンドファイルを一つに結合してください。

2) 2番目の間接コマンドファイルを、 1番目の間接コマンドファイル中ではなく386 | LIB上でオープンしてください。

Fatal Error: Unknown command switch -- "XXX"

原因：　386 | LIB のコマンドスイッチでないものがコマンドライン上に指定されたため。

処置：　スイッチ名の綴りが正式な386 | LIBスイッチと合っているかどうかチェックしてください。

索引

386 | ASM TOOL KIT　ユーザーズマニュアル
第4部　ライブラリアン 386 | LIB

英数・記号

あ行

か行

た行

は行

ま行

ら行

FMTOWNS

386 | ASM TOOL KIT
ユーザーズマニュアル
81SP-0073-1-0

発 行 日　1989年10月
発行責任　富士通株式会社

Printed in Japan